The OMAC API Open Architecture Methodology

EXECUTIVE SUMMARY

Open modular architecture controllers technology offers great potential for integration of process improvements and better satisfaction of process requirements. With an open architecture, controllers can be built from best value components from best in class services. The need for open-architecture controllers is high, but vendors are slow to respond. One reason for the delay in industry action is that no clear open-architecture solution has evolved. In an effort to promote open architecture control solutions, a workgroup within the Open Modular Architecture Controller (OMAC) users group is working on defining an OMAC Application Programming Interface (API). The goal of the OMAC API workgroup is to specify standard APIs for a set of open architecture controller components. This document contains background information, design methodology and actual API definitions.

As background, the following material will be presented:

- OMAC API definition of open architecture
- advantages and impediments to open architectures
- overview of the OMAC API reference model.

At a high level of conceptual design, the OMAC API reference model will be presented and includes the following items:

- OMAC API core modules
- application framework
- application design and examples.

The OMAC API reference model does not specificy a reference architecture. Instead, modules can be freely connected. In lieu of a reference architecture, the document includes several reference examples.

At a detailed level of design, the OMAC API specification methodology will be presented and subscribes to the following principles:

- API programming abstraction is used
- Object Oriented techniques for encapsulation, inheritence, specialization and object interaction are applied
- Client/Server is the communication model
- Proxy Agents provide transparency of distributed communication
- Finite State Machine (FSM) is the behavior model
- Finite State Machine (FSM) are passed as data to then provide control
- Reusability of software components is achieved through foundation classes

- System objects are mirrored in human machine interface
- No specification of an infrastructure is attempted instead a commitment to a *Platform + Operating System + Compiler + Loader + Infrastructure suite* is necessary for it to be possible to swap modules.

1 BACKGROUND

Most Computer Numerical Control (CNC) motion and discrete control applications incur high cross-vendor integration costs and vendor-specific training. On the other hand, in a modular, standard-based, open-architecture controller modules can be added, replaced, reconfigured, or extended based on the functionality and performance required. Modifications to a module should provide equivalent or better functionality as well as offer different performance levels. Ideally, the module interfaces should be vendor-neutral, plug-compatible and platform independent.

However, it is important to note that openness alone does not achieve plug-and-play. One vendor's idea of openness need not be the same as another vendor's. Openness is but one step towards plug-and-play. In reality, plug-and-play openness is dependent on a standard. This leads to the following definition of an open architecture controller:

An open architecture control system is defined and qualified by its ability to satisfy the following requirements:

Open provides ability to piece together systems from components, ability to modify the way a controller performs certain actions, and ability to start small and upgrade as a system grows.

Modular refers to the ability of controls users and system integrators to purchase and replace controller modules without unduly affecting the rest of the controller, or requiring extended integration engineering effort.

Extensible refers to the ability of sophisticated users and third parties to incrementally add functionality to a module without completely replacing it.

Portable refers to the ease with which a module can run on different platforms.

Scalable allows different performance levels and size based on the platform selection. Scalability means that a controller may be implemented as easily and efficiently by systems integrators on a stand-alone PC, or as a distributed multi-processor system to meet specific application needs.

Maintainable supports robust plant floor operation (maximum uptime), expeditious repair (minimal downtime), and easy maintenance (extensive support from controller suppliers, small spare part inventory, integrated self-diagnostic and help functions.)

Economical allows the controller of manufacturing equipment and systems to achieve low life cycle cost.

Standard Interfaces allow the integration of off-the-shelf hardware and software components and a standard computing environment to build a controller. Standard interfaces are vital to plug-and-play.

Degree of openness can be evaluated by comparing a claim of openness against the above requirements. Herein, the concept of an open-architecture control system that supports openness, and the auxiliary requirements will be identified as *"open, openness or open architecture."*

No approval or endorsement of any commercial product by the authors or their employers is intended or implied. Certain commercial equipment, or materials are identified in this report in order to facilitate understanding. Such identification does not imply recommendation or endorsement by the authors or their employers, nor does it imply that the materials or equipment identified are necessarily the best available for the purpose.

1.1 Advantages of Open Architecture Technology

Based on specific instances of problems encountered by users of proprietary controllers, the following list of open-architecture requirements was generated. An open architecture should be able to do the following:

- provide a migration path from existing practices;
- allow an integrator/end user to add, replace, and reconfigure modules;
- provide the ability to modify spindle speed and feed rate according to some user-defined process control strategy;
- allow access to the real-time data at a predictable rate up to the servo loop rate;
- allow full 3-D spatial error correction using a user-defined correction strategy;
- decouple user interface software and control software and make control data available for presentation;
- provide communication functions to integrate the controller with other intelligent devices;
- increase the ability for 3rd party software enhancements. Examples of 3rd party enhancements include:
  - replace a PID control law with a more sophisticated Fuzzy Logic control law
  - collect servo response data with a 3rd party tool, and set tuning parameters in the appropriate control law
  - add a force sensor, and modify the feed rate according to a user defined process model
  - perform high resolution straightness correction on any axis
  - replace the user interface with a 3rd party user interface that emulates a user interface familiar to your machine operators.

The initial validation strategy for the OMAC API would be to insure that this list of capabilities can be addressed.

1.2 Impediments to Open Architecture Technology

It is difficult to define a specification that is safe, cost-effective, and supports real-time performance.

A specification must factor in current practices, as well as anticipate evolving technologies. To be successful, the open architecture definition must be implementable with current computer technology and skills. Further, an open architecture specification cannot be so rigidly defined as to preclude future technology upgrades. An open architecture specification must be able to grow.

Of great importance within the controls domain is the requirement for guaranteed, hard-real-time performance. Without this, safety is at risk. Safety is a major concern voiced within the controller industry which is especially concerned with the issues of liability and allocation of responsibility within an open architecture paradigm. New industry practices would have to be adopted for open architecture controllers. A greater responsibility would be placed on the integrator. Conformance testing would play a larger role. Conformance could require regression and boot-up testing and verification procedures to guarantee proper operation.

A further hindrance is the fact that modules are not "self-contained." Defining an infrastructure within which the modules can operate is necessary and quite difficult. We consider the *infrastructure* to be defined as the services that tie the modules together and allow modules to use platform services. The infrastructure is intended to hide specific hardware and platform dependence; however, this is often difficult to achieve.

Containing the scope of the specification is also difficult. Openness goes beyond run-time APIs. There can be "other" APIs, including configuration, integration, and initialization. As an example, consider the simple use of a math library API. Even there, specification of the math library implementation must be done to select either a floating point processor or software emulation.

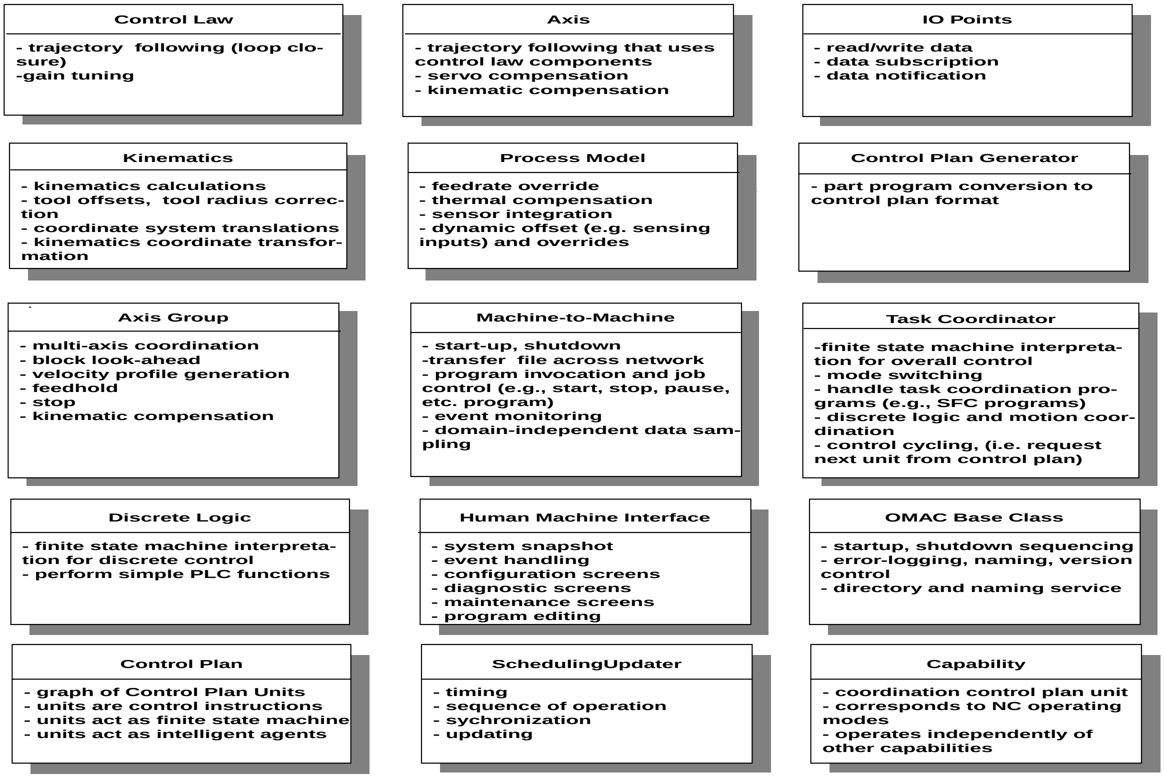

Figure 1: OMAC API Core Modules

Finally, group and industry dynamics can be a problem. From a workgroup perspective, getting people to agree can be a challenge because there are difficult trade-offs in modularization, scope-covered,life cycle benefits to be realized, costs, time to market, and complexity. It is recognized that industry will find it difficult to adopt the OMAC paradigm, due to entrenchment in the legacy of prior implementations, the "comfort zone" of past practice and culture, the investment hurdle to effect change, and the shortage of skilled resources. Proper acculturation, training and education of people and an orderly introduction, demonstration, robustization, motivation, and scale-up will be needed to realize the potential benefits. From an industry perspective, many companies do not perceive any direct benefit from an open architecture. Overcoming the workgroup inertia and industry skepticism by promoting and articulating the benefits of open architecture remains a fundamental key to open architecture acceptance.

2 REFERENCE MODEL

The OMAC API requirements were derived from the OMAC or "Open Modular Architecture Controller" requirements document [OMA94]. The OMAC document describes the problem with the current state of controller technology and prescribes open modular architectures as a solution to these problems. OMAC defines an open architecture environment to include Platform, Infrastructure, and Core Modules. ‘

OMAC API defines a *module* to have the following characteristics:

- significant piece of software used in composing controller
- grouping of similar classes
- well-defined API
- well-define states and state transitions

- replaceable by any piece of software that implements the API, state and state transitions.

Using the OMAC specification model as a baseline, Figure 1 diagrams the OMAC API Core Modules including a brief description of a module's general functional requirements. The Core Modules have the following general responsibilities:

Task Coordinator modules are responsible for sequencing operations and coordinating the various motion, sensing, and event-driven control processes. The task coordinator can be considered a finite state machine (FSM) accepting directives one at a time from an operator or as a stored sequence of instructions in the form of a Control Plan.

Control Plan Generator modules are responsible for translating the part program into a Control Plan. Similarly, translations for IEC 1131-3[IEC93] programs and other formats are responsible for producing Control Plans.

Axis Group modules are responsible for coordinating the motions of individual axes, transforming an incoming motion segment specification into a sequence of equi-time-spaced setpoints for the coordinated axes.

Axis modules are responsible for servo control of axis motion, transforming incoming motion setpoints into setpoints for the corresponding actuators.

Kinematics Models modules are responsible for kinematic transformations including geometric correction, tool offsets, and effects of tool wear. Computing forward and inverse kinematics, mapping and translating between different coordinate systems, and resolving redundant kinematic solutions are examples of kinematic model functionality.

Control Law components are responsible for servo control loop calculations to reach the specified setpoints.

Human Machine Interface or HMI modules are responsible for remotely handling data, command, and event service of an internal controller module. Defining a presentation style (e.g., GUI look and feel, or pendant keyboard) is not part of OMAC API effort.

Process Model is a component that contains dynamic data models to be integrated with the control system. Process control components not specified in this architecture produces adjustments or corrections to nominal rates and path geometry in the form of this component. Feedrate override and thermal compensation are examples of process model functionality. The process model is important to the concept of extensible open systems.

Discrete Logic modules are responsible for implementing discrete control logic or rules that can be characterized by a Boolean function from input and internal state variables to output and internal state variables. More than one discrete logic module is permitted, but not necessary. Multiple discrete logic modules is similar to having many PLC's networked together within the same computing platform.

I/O Points are responsible for the reading of input devices and writing of output devices through a generic read/write interface. The goal is to provide an abstraction for the device driver. Logically related IO may be clustered within a Discrete Logic module.

Scheduling Updater is a module that provides centralized scheduling functionality, that includes, timing, synchronization and sequencing. This mechanism is provided since most real-time operating systems do not explicitly provide sequences of periodic updating.

Control Plan is an aggregation of classes that form the basis of control and data flow within the system. A Control Plan Unit is a base class that contains instructions for a module. A Control Plan consists of a graph of Control Plan Units. *Motion Segment* is a derived class of Control Plan Units for motion control. *Discrete Logic Unit* is a derived class of Control Plan Units for discrete logic control. A Control Plan Unit could be a Task Coordinator.

OMAC Base Class provides a uniform API base class for an OMAC module. The OMAC base class defines a state model and methods for start-up and shutdown. The OMAC Base Class defines a uniform name and type declaration and provides an error-logging interface. The OMAC Base Class maintains a global directory service for name lookup and reference binding.

Capability is an object to which the Task Coordinator delegates for specific modes of operation. Capability corresponds to traditional operating modes (AUTO, MANUAL, MDI, etc.) At Capability Level, there is no coordination between Capabilities. The difference between a Capability and a Control Plan Unit is subtle with the distinction between the two that a Capability is tightly coupled to a Task Coordinator module. In fact, a Capability or a Control Plan Unit could itself be a Task Coordinator. This interchange is possible since all are finite state machines.

Machine-to-Machine modules are responsible for connecting and communicating to controllers across different domains (address spaces). An example of this functionality is the communication from a Shop Floor controller to an individual machine controller on the floor.

Some clarifying observations about modules include:

- Interchangeable modules may differ in their performance levels.
- Modules may provide more functionality (added value) than required in the specification. *Specialization* of a module interfaces is the mechanism to achieve additional functionality.
- A controller may have more than one instance of a module.
- Modules can be explicitly control-related (e.g., Axis, Axis Group) or be service-related (e.g., OMAC Base Class or Scheduling Updater) for capturing common functionality that is inherited.
- Modules do not need to run as separate threads (or intelligent agents.) Systems can be built from a single thread of execution.
- Modules can contain multiple threads of execution.
- Modules may be used to build other components. For example, a discrete mechanism, such as a tool changer component, can be built using the core OMAC modules.
- Multiple instances of a module are required to handle different configurations. For example, assume a system with 3 axis x,y,z and a spindle. One would create three Axes Group objects at configuration time, `ag1, ag2, ag3`, with the following configuration:

  ```
  ag1: x,y,z
  ag2: spindle
  ag3: x,y,z, spindle
  ```

  For most machining where the motion control and the spindle are loosely related, a reference to `ag1, and ag2` would be used. However to do a Rigid Tap requiring tight synchronization of the spindle and motion, a reference to `ag3` would be used.

2.1 Reference Architecture

In the interest of flexibility, scalability, and reusability, OMAC API does not specify a fixed architecture. Instead, OMAC API specifies API for components to support the OMAC core modules. At a higher level, the assembly of the OMAC modules into a system requires an integration architecture and an assembly strategy described below for connecting modules. Suggestions are offered, but are not mandated.

OMAC API assumes a module assembly described by this abstraction hierarchy:

- Foundation Classes

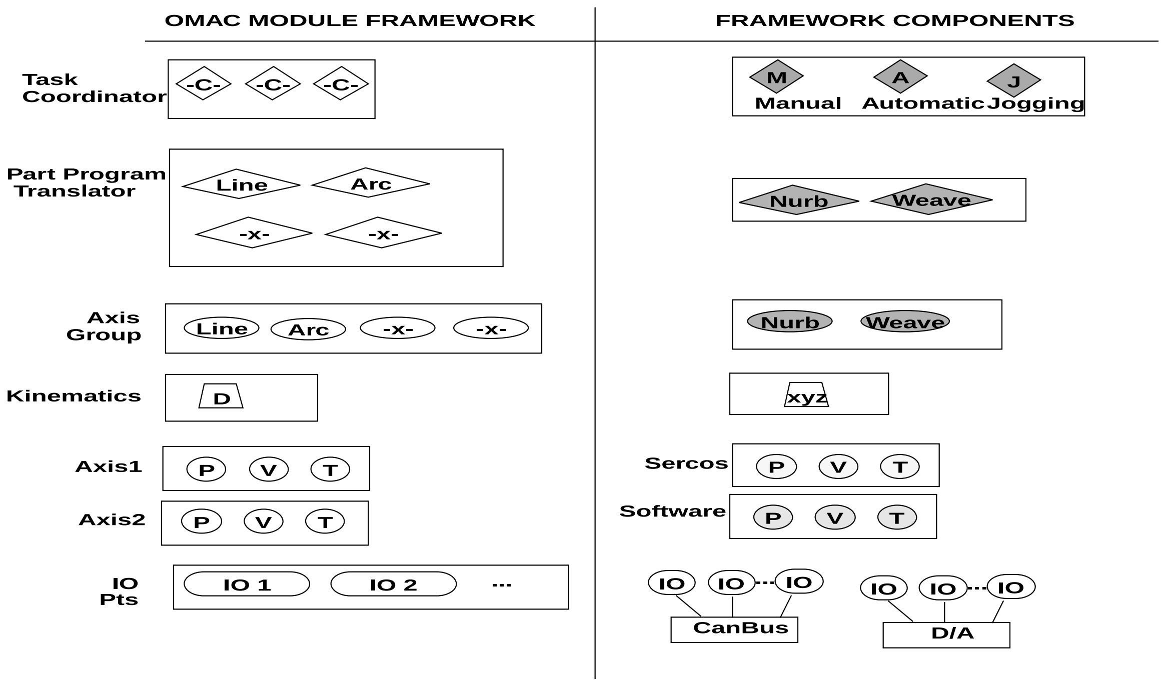

Figure 2: Control Framework

- Framework Components
- Core Modules
- Integration Architecture
- Application Architecture

The foundation classes are the building blocks that may be found in multiple modules. For example, the class definition of a point would be found in most modules. Framework components are instances of foundation classes that can be integrated into the core modules. For example, LinearPath or CircularPath objects are framework components of a Control Plan Unit for motion. The core modules have the functionality as previously outlined. An integration architecture describes a configuration methodology for component topology, timing, and inter-component communication protocols. An application architecture specifies components and interconnections selected for a particular application, from the choices allowed by the generic or reference integration architecture. With the application architecture, users can develop and run programs. Some candidate distributed reference architectures include the following: agent-based, DCOM [DCO], CORBA [COR91], RCS [Alb91], OSACA [OSA96], or EMC [PM93].

2.2 Application Framework

In the OMAC API, an application control systems is built as a set of connected modules that use other module services through the published API. The OMAC API specifies module APIs aimed for the system integrator. At this level, the system integrator links ".o" object files (or linked libraries) to assemble a controller. The .o's correspond to procured modules bought as commercial off-the-shelf technology (COTS). The assembly of OMAC API modules in such a manner is referred to as the *framework paradigm*.

Object-oriented *frameworks* are sets of prefabricated software and building blocks that are extensible and can be integrated to execute well-defined sets of computing behavior. Frameworks are not simply collections of classes. Rather, frameworks come with rich functionality and strong "pre-wired" interconnections between the object classes.

This contrasts with the procedural approach where there is difficulty extending and specializing functionality; difficulty in factoring out common functionality; difficulty in reusing functionality that results in duplication of effort; and difficulty in maintaining the non-encapsulated functionality. With frameworks, application developers do not have to start over each time. Instead, frameworks are built from a collection of objects, so both the design and the code of a framework may be reused.

In the OMAC API framework the prefabricated building blocks are the COTS implementations of the OMAC modules and framework components. As a simple example, Figure 2 illustrates a framework for a typical controller application. An application developer buys the modules, and then the application developer "puts the pieces together."

Within the example, there is a task coordinator module which has containers for inserting capabilities (in the figure represented by a -C- framed by a diamond). The capabilities include Manual, Automatic or Jogging. The application developer is free to put one or more of these capabilities into the task coordinator or develop a unique capability. For Control Plan Generator and Axis Group, the application developer is already provided line and arc path descriptions but can plug in Nurb (Non-Uniform Rational B-Spline) or Weave path descriptions. Once again, application developers could uniquely develop a path description. For the Axis modules, the application developer has the possibility to do position (P), velocity (V) or torque (T) control in software, hardware or some combination of hardware and software. For software P control, the application developer would select a control law object from the Software set. For hardware P control, the application developer would select a control law object from the Sercos set.

Using the OMAC API framework paradigm, application development involves three groups:

Users define the behavior requirements and the available resources. Resources include such items as hardware, control and manufacturing devices, and computing platforms. For behavior, the user defines the performance and functionality expected of the controller. Performance includes such characteristics as how fast or how accurate the application must be. Functionality defines the controller capability such as the ability to handle planar part features versus complex part features.

System Integrators select modules and framework components to match the application performance and functional requirements. The system integrator configures the modules to match the application specification. The system integrator uses an integration architecture to connect the selected modules and verifies the system operation. The system integrator also checks compliance of modules to validate the user-specification of performance and timing requirements.

Control Component Vendors provide module and framework component products and support. For control vendors to conform to an open architecture specification, they would be required to conform to several specifications including the following:

- customer specifications
- module class specification
- system service specification

The system service describes the platform and infrastructure support (such as communication mechanisms) and the resources (disks, extra memory, among others) available. Computer boards have a device profile that includes CPU type, CPU characteristics and the CPU performance characteristics. Included within the profile is the operating system support for the CPU. A spec sheet or computing profile [SOS94] is required to describe the system service specification that would include such areas as platform capability, control devices, and support software.

2.3 Application Design and Examples

A system design is divided into two phases. The first phase is *Architectural Design* and deals with the decomposition of the system into subsystems (i.e., OMAC modules). This design activity corresponds to the previous discussion on the OMAC Reference Architecture.

The second phase is called *Detailed Design* and is responsible for detailing individual object API, that is, the object attributes and methods. At this phase, one determines which objects are available, the extent of object capabilities, and whether the objects need to be bought or built. This phase corresponds to the discussion on putting a system together with the OMAC Framework.

Since there is no explicit OMAC reference architecture, how to compose a system architecture from OMAC modules is left to the developer. This offers much flexibility, but without guidance, can be confusing. This section will give some application architecture examples for clarification. This section starts with a simple applications and then develops a series of examples to illustrate the stages of development one might encounter when building an application architecture. The examples will highlight the relations between OMAC modules (as opposed to the data flow.) However, one can assume that the architectures are hierarchical and directives flow from top to bottom.

2.3.1 Operator Control of a Set of IO Points Example

The simplest case is an operator controlling several IO points. The OMAC API model allows the connection of an Human Machine Interface (HMI) object to several IO points. Figure 3 shows the simple connection between HMI and IO points. The rationale for such a simple example is to show that the OMAC API is not monolithic, and one can put a small system together. With this ability, OMAC systems can start small and be pieced together.

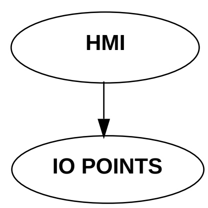

Figure 3: Operator Control of a Set of IO Points

2.3.2 One Axis Bootstrap

After establishing an HMI and IO connection, the natural progression in building an NC machine tool controller is to add an axis of motion under manual control. This scenario is typical in offline assembly and testing of an axis that may eventually be assembled in a multi-axis NC machine tool. Jogging and Homing are the primary functionality used. At this point, there is no coordination with any other motion, mechanism, or state in the NC machine tool. During this stage of the assembly of a machine tool, it is also helpful to perform the calibration, tuning, or health monitoring tests.

Assume that the Axis module will consist of a PWM motor drive, an amplifier enable control, an amplifier fault status signal, an A-QUAD-B encoder with marker pulse and switches for home and axis limits. Figure 4 shows a one-axis system that uses two Control Laws, one for PID control of Position, and another to do PID control of velocity. The Axis will be outputing accelerations to the actuator and reading encoder values through IO points referenced in the Axis module. For operator control of the axis, an HMI module mirrors exiss for the Axis module as well as mirrors for each Control Law module. The mirrors provide a snapshot of control system objects and use proxy agents for communication.

2.3.3 Programmable Logic Example

Next, let's consider a case of work-handling equipment that provides peripheral functions for an NC machine tool. The equipment includes two hydraulically actuated, two-position on-off mechanisms, named, Loader and Unloader. Let their sensing, actuation, and control be under a Discrete Logic module, named `LUNL` whose sequence of operations was originally specified in some manner conforming to IEC 1131-3, and subsequently translated into a Control Plan Unit, named `CPlunl`.

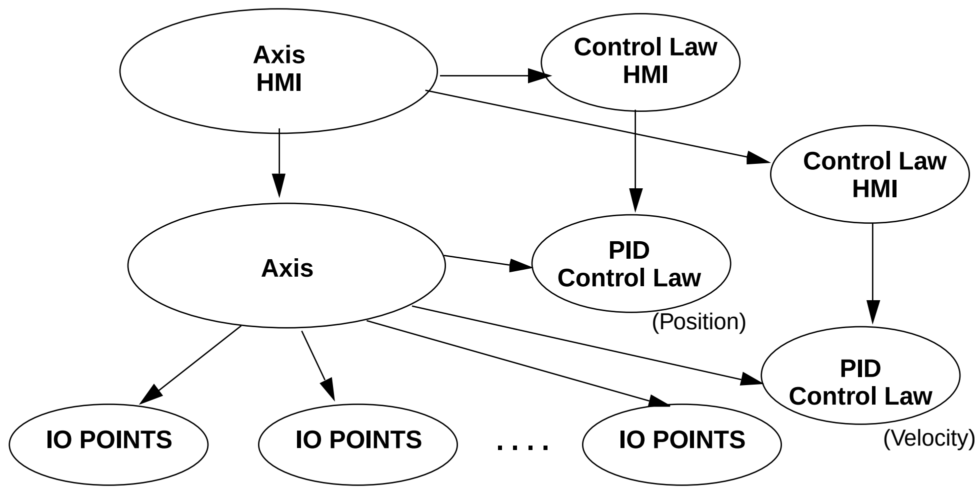

Figure 4: One Axis Bootstrap

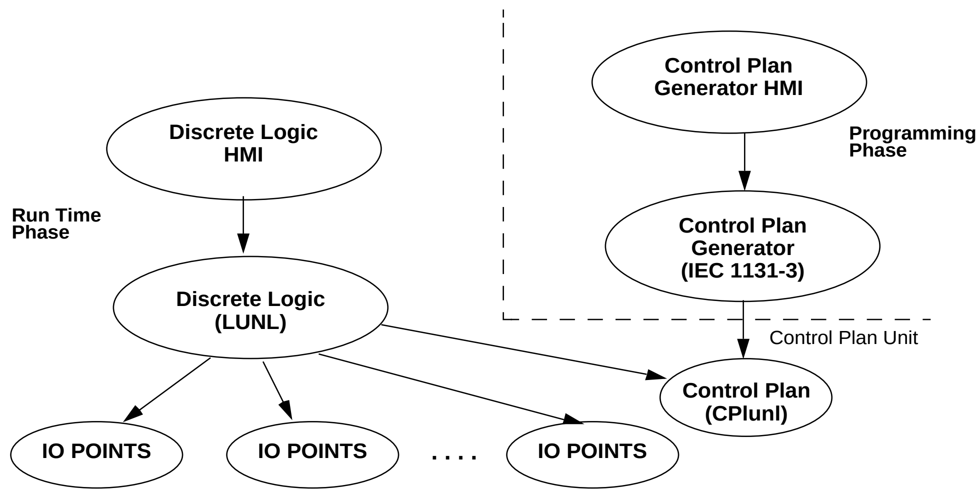

Figure 5: Loader/Unloader Discrete Logic Control

Figure 5 illustrates the relationship of different OMAC modules within this LUNL application. Within the block diagram, two phases, Programming Phase and Run Tim Phase, are shown. However, there are also other phases to be considered.c The following steps sketch the different phases of system development.

1. In the Programming phase,
   a. Given the IEC 1131-3 code, perform logical mapping onto IO and functions
   b. Generate number of Control Plan Units (FSM), with one associated with each state. (Appendix A contains a hypothetical FSM in table format.)
   c. Group Control Plan Units to become a LUNL Control Plan
2. At configuration phase,
   a. Perform physical mapping IO and functions
   b. Load Control Plan into the Discrete Logic Module
3. At initialization phase,

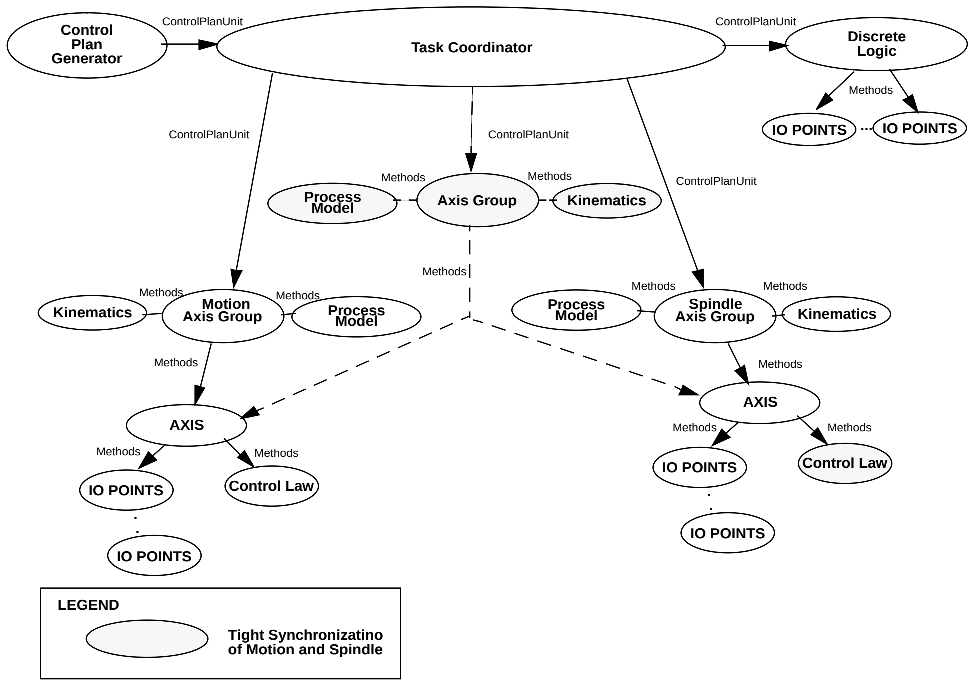

Figure 6: Drilling Example

   a. Resolve external object and module references
   b. Register events

4. At runtime phase,
   a. Clients (HMI or IO Points) generate events
   b. LUNL FSM execution at Discrete Logic Module scan rate interprets each Control Plan Unit which is an FSM.

2.3.4 Drilling Motion Control Example

To keep matters simple, an example describing programmed NC for one-axis drilling will be developed. A typical one-axis drilling workstation would perform some holeworking operations, e.g., drilling with a spindle drill-head, boring a precision bore, counter-boring the bored hole, probing the (axial) location of the counterbored shoulder.

Figure 6 illustrates the module and component relationships for a drilling application. For drilling the Axis motion control also requires coordinated motion supplied by an AxisGroup module. Another Axis module is required for Spindle control. Spindle drive components are assumed to provide a facility for setting spindle speed and direction and to start and stop spindle rotation. Another Axis Group to control the Spindle is required, and one for controller both the Motion Axis and the Spindle Axis (shown as shaded with dashed line connections). Generally, the Spindle Axis will not need a Control Law, however, when it is synchronized with motion it will require servoed control.

In the diagram, a Task Coordinator exists to provide program control. A ControlPlanGenerator module translates a part program into ControlPlanUnits. The primary command communication between modules is reflected in the digrams by showing the keyword “Method” or “ControlPlanUnits” (which uses a method to pass it) next to an arrow. A Discrete Logic Module, typical of the previous example, exists as an equivalent for part loading and unloading, as well as machine state (e.g., temperature, etop). To improve predictability and reduce variation, a Process Model module will exist to integrate sensing and control to detect tool breakage by monitoring spindle torques and thrust forces. A simple Kinematics module exists to model the workspace and handle different tool offsets and part placements.

3 SPECIFICATION METHODOLOGY

To satisfy the OMAC open architecture specification, a standard API for each of the Core Modules would be defined. Consequently, the primary goal of the OMAC API workgroup is to define standard API for the Core Modules. This section will refine the concept of "API" and describe the OMAC API specification methodology. The API specification methodology applies the following principles:

- Stay at API level of specification. Use IDL to define interfaces.
- Do not specify an infrastructure.
- Use Object Oriented technology.
- Use general Client Server communication model, but use state-graph to model state behavior.
- Use Proxy Agents to hide distributed communication.
- Finite State Machine (FSM) is model for data and control.
- Define Foundation Classes to foster the concept of reusable assets.
- Mirror system objects in human machine interface.

The following sections will discuss these principles.

3.1 API Specification

API stands for Application Programming Interface, and refers to the programming front-end to a conceptual black box. The math function "`double cos(x)`" specifies the function name, calling sequence, and return parameter, not how the cosine is implemented, be it table lookup or Taylor series. Of importance to the API specification is the function *signature* and its calling and return sequence, assuming of course, that cosine doesn't take too long. Behavior is an explicit element within the API definition and relies on a defined state transition model. A (standard) API is helpful because programming complexity is reduced when one alternative exists as opposed to several. For example, the cosine signature is generally accepted as `cos(x)`, not `cosine(x)`. This is a small but significant standardization.

At a programmatic level, the importance of a standard API can be seen within the Next Generation Inspection Project (NGIS) at NIST[NGI]. The NGIS project has integrated three commercial sensors and one generic sensor into the Coordinate Measuring Machine controller. Taming diversity was a problem. Each sensor had a different "front-end" - one had a Dynamically Linked Library (.DLL) interface, one had a memory mapped interface, one had a combination port and memory mapping. None of the sensors had the same API. Yet, all of the sensors were "open."

There exists a problem selecting the API specification language. The specification language must be flexible enough to support a variety of implementation languages and platforms. OMAC API chose IDL, or the Interface Definition Language, for its specification language [COR91]. IDL is a technology-independent syntax for describing interfaces. In IDL, interfaces have attributes (data) and operation signatures (methods). IDL supports most object-oriented concepts including inheritance. IDL translates to object-oriented (such as C++ and JAVA) as well as non-object-oriented languages (such as C). IDL specifications are compiled into header files and stub programs for direct use by application developers. The mapping from IDL to any programming language could potentially be supported, with mappings to C, C++, and JAVA available.

3.2 Object Oriented Technology

OMAC API uses an object-oriented (OO) approach to specify the modules' API with class definitions. The following terms will define key object-oriented concepts. A *class* is defined as an abstract description of the data and behavior of a collection of similar objects. Class definitions aggregate both data and methods to offer *encapsulation*. An *object* is defined as an instantiation of a class. For example, the class `SERCOS-Driven Axis` describes objects in the running machine controller. A 3-axis mill would have three instantiations of that class – the three objects described by that class. An *object-oriented program* is considered a collection of objects interacting through a set of published APIs. A by-product of an object-oriented approach is *data abstraction* which is an effective technique for extending base types to meet the programmer needs. A "complex number" data abstraction, for example, is certainly more convenient than manipulating two doubles.

3.2.1 Inheritance

Inheritance is useful for augmenting data abstraction. OO classes can inherit the data and methods of another class through class derivation. The original class is known as the *base or supertype class* and the class derivation is known as a *derived or subtype class*. The derived class can add to or customize the features of the class to produce either a specialization or an augmentation of the base class type, or simply to reuse the implementation of the base class. To achieve a framework strategy, all OMAC API class signatures (methods) are considered "virtual functions." Virtual functions allow derived classes to provide alternative versions for a base class method.

Using an Axis module as a server, assume that all the axis does is set a variable x.

```
class Axis
{
  virtual void set_x(float x);
private:
  double myx;
}

application()
{
  Axis ax1;
  ax1.set_x(10.0);
}
```

To extend the server, a base class to add an offset to its value before each set is derived. This could also be achieved on the server side if so desired.

```
class myAxis : public Axis
{
    virtual void set_x(float x){ x= x + offset; Axis::set_x(x); }
  private:
    double myx;
    double offset; // set elsewhere for offset calculation
}

application()
{
    Axis ax1;
```

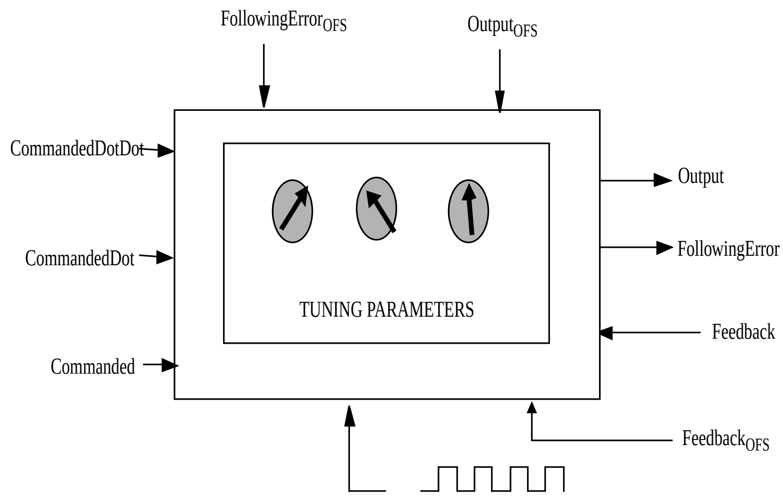

Figure 7: General Control Law

```
    myAxis ax2;
    double val;
    double offset;

    val=10.0;
    ax1.set_x(val+offset); // explicit offset in application code
    ax2.set_x(val);        // offset hidden by configuration
}
```

3.2.2 Specialization

OMAC API leverages the OO concept of inheritance to use base and derived classes to add specialization. When defining a control law, one has many options including PID, then Fuzzy, Neural Nets, and Nonlinear. This plethora of options implies a need to contain the realm of possibilities. The OMAC API approach is to define a base type (generally corresponding to one of the OMAC Core Modules) and then add specialized classes.

The control law module illustrates the base and derived class specialization. The responsibility of the Control Law module is conceptually simple – use closed loop control to cause a measured feedback variable to track a commanded setpoint value using an actuator.

Figure 7 illustrates the definition of a base control law. The concept of tuning is encapsulated within the black box and is conceptually controlled via "knob turning." The concept of accepting third party signal injection is handled by the inclusion of pre-and post-offsets (or injection points). These offsets allow sensors or other process-related functionality to "tap" and dynamically modify behavior by applying some coordinate space transformation. The IDL definition of the illustrated control law module follows. The IDL keyword `interface` signifies the start of a new interface, corresponding to a C++ class.

```
interface CONTROL_LAW
{  // Parameters
  void set_commanded(double setpoint);
  double get_commanded();

  void set_commanded_dot(double setpointdot);
  double get_commanded_dot();
```

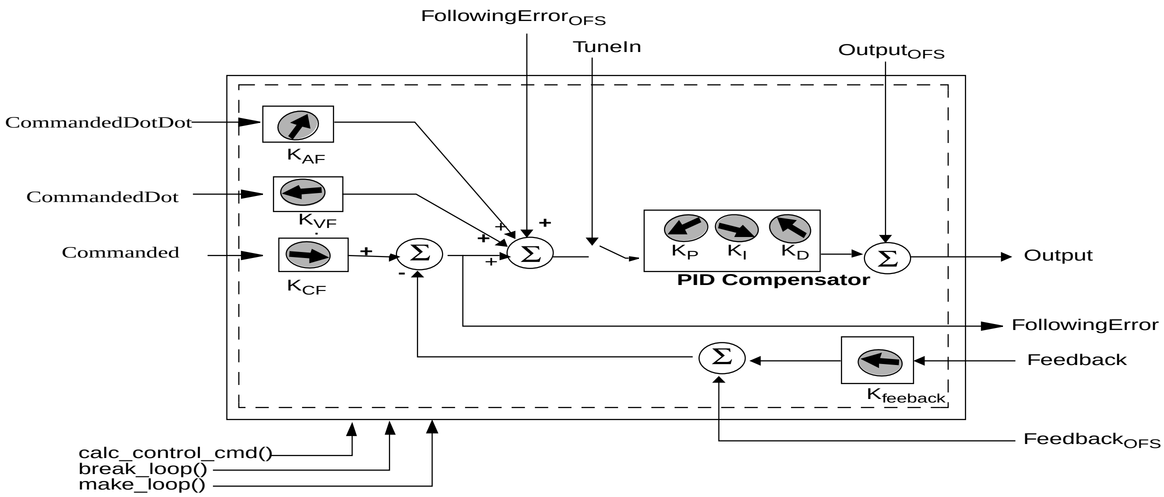

Figure 8: PID Control Law

```
  void set_commanded_dot_dot(double setpointdotdot);
  double get_commanded_dot_dot();

  void set_output(double value);
  double get_output();

  void set_feedback(double actual);
  double get_feedback();

  void set_following_error(double epsilon);
  double get_following_error();

  // Offsets
  void set_following_error_offset(double preoffset);
  double get_following_error_offset();

  void set_output_offset(double postoffset);
  double get_output_offset();

  void set_feedback_offset(double postoffset);
  double get_feedback_offset();

  void set_tune_in(double value); // enable with break_loop
  double get_tune_in();
};
```

Each CONTROL_LAW specialization is a subtype whereby each subtype inherits the definition of the supertype. By applying this concept, an evolutionary process evolves to adapt to changes in the technology. At first, only highly-demanded subtypes, such as PID, were handled. Figure 8 conceptually illustrates the PID specialization of the control law. The IDL definition of the PID control law follows.

```
interface PID_TUNING: CONTROL_LAW
{ // Attributes
  double get_Kp();
  double get_Ki();
  double get_Kd();

  void set_Kp(double val);
  void set_Ki(double val);
  void set_Kd(double val);

   double get_Kcommanded();
   double get_kcommanded_dot();
   double get_Kcommanded_dot_dot();
   double get_Kfeedback();

   void set_Kcommanded(double val);
   void set_kcommanded_dot(double val);
   void set_Kcommanded_dot_dot(double val);
   void set_Kfeedback(double val);
};
```

OMAC API also uses inheritance to maintain levels of complexity. Level 0 would constitute base functionality seen in current practice. Level 2 would constitute functionality expected of advanced practices. Level 3, 4,..., n would constitute advanced capability seen in emerging technology, but unnecessary for simple applications.

3.3 Client Server Behavior Model

OMAC API adopts a client server model of inter-module communication. In the client/server model, a module is a *server* and a user of a module is called a *client*. Modules can act as both a client and a server and cooperate by having clients issue requests to the servers. The server responds to client requests. A client invokes *class methods* to achieve behavior. A client uses *accessor methods* to manipulate data. Accessor methods hide the data physical implementation from the abstract data representation. The server reacts to the method invocation and performs the corresponding method implementation and sends a reply (either an answer or a status) back to the client.

As a server, a module services requests from clients that can be immediately satisfied or that may require multiple cycles. Multiple cycle service requests require state space logic to coordinate the interaction. OMAC API define three types of service requests: (1) parametric requests, (2) command requests and (3) updating requests.

Parametric service requests are generally the get/set methods and are, in theory, immediately satisfied. They do not require state space logic.

Command service requests are command methods which, assuming a differing subsequent command, causes a change in the server's state space (or state transition) and results in a new server state. These command requests may run one or many axis cycles - such as `move_to()` absolute position. Repeated cycles of the same command methods require a state transition mechanism for coordination between the client and the server. Service requests require an FSM to coordinate the client server interaction.

Updating service requests coordinate the execution of a module, for example, `processServoLoop()` or `update()` for Axis module. The `processServoLoop` function provides cyclic execution - e.g., axis module is executed once per servo loop period. In this mode, the axis software would be running as a data flow machine: at every period, it accesses the data (e.g., commanded position, actual feedback) and derives a new setpoint.

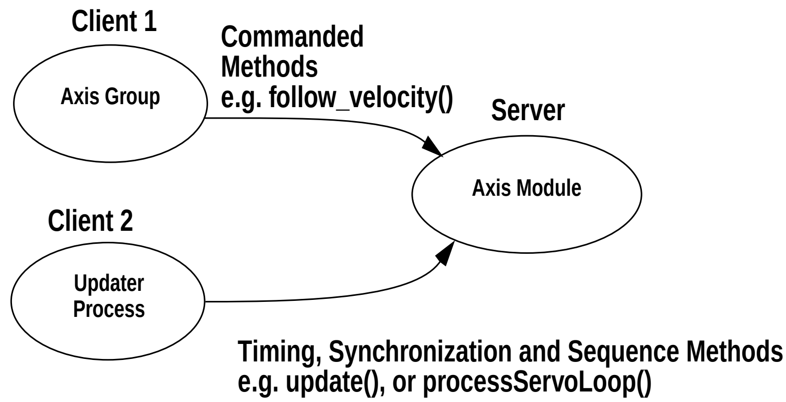

Figure 9: Multiple Threads of Control

If an OMAC module is periodic, it may derive the method `update()` by inheriting it from the Scheduling Updater class `Updatable`. For Axis, the method `update()` is a wrapper that calls `processServoLoop`. The `update()` method simplifies invocation for the `updater` since all modules have the same signature, the `updater` can go down a list of modules and invoke one signature.

Client Command and Updating service requests may come from separate threads of control. Figure 9 illustrates a server with multiple clients running in two separate processes: an Axis Group process for issuing setpoints and an Periodic Updater process to coordinate execution. (These processes may be running in one or more threads.) Generally, the Commanded service requests would come from an Axis Group module that is issuing setpoints to multiple axes. A Scheduling Updater module running in another thread of execution provides timing, synchronization and sequencing service for the Axis module. This Scheduling Updater module may be tied to some hardware device (such as a timer) to guarantee periodic execution behavior.

An example to illustrate the nuances of this multi-client/server interaction will be developed. First, the object naming and constructor definition that is done at configuration time will be sketched. The integration creates object references (i.e., `io1`, `io2`, `ax1`, `axgrp1`) and then binds addresses to the created objects through some name registration. Since `ax1` and `axgrp1` are periodic updating OMAC modules, they have inherited a method `update()` and register with the Periodic_updater `updater` using its `register_updatable()` method. The second parameter field in `register_updatable()` method is the clock divisor.

```
integration_process_init(){
    // initialize parameters
    Periodic_Updater updater;

    IOpoint  io1= new IOpoint("encoder1");
    IOpoint  io2= new IOpoint("actuator1");}
    Axis ax1= Axis("Axis1", io1, io2);
    AxisGroup axgrp1= AxisGroup("AxisGroup1", ax1);

    updater.set_timing_interval(.01);  // 10 millisecond period
    updater.register_updatable((Updatable *) axgrp, 2);
    updater.register_updatable((Updatable *) ax1, 1);
}
```

Next, a sequence of operations will highlight the connection between the Scheduling Updater (`Updater`), the Axis Group module (`AxGrp`), the Axis module(`Axis`) and the actuator and encoder IO points. Within the `Axis` module, references to the component classes `Axis_velocity_servo, Axis_Command_Output` and `Control Law` module will be made. (Readers are referred to Section 4.0 to further review Axis components.)

In lieu of an Object Interaction Diagram, the following pseudo code tracks the sequence of operations to set `ax1` to follow velocity profile, send a commanded velocity, read the axis actual velocity, compute the next acceleration setpoint using a Control Law and then output a commanded acceleration to the IO. Indentation indicate levels of descent in the calling stack.

```
updater→axgrp1→update()
    axgrp1→ax1→following_velocity()
            ax1→Axis_velocity_servo→start_velocity_following()
    axgrp1→set_commanded_velocity(commandedvelocity)
updater→ax1→update()
        ax1→processServoLoop()
        Axis_velocity_servo→velocity_update_action()
        Axis_velocity_servo→Axis_Command_Output→get_velocity_command()
        Axis_velocity_servo→Axis_Sensed_state→get_actual_velocity()
            io1→get();
        Axis_velocity_servo→ControlLaw→(load parameters)
        Axis_velocity_servo→ControlLaw→calc_control_cmd
        Axis_velocity_servo→ControlLaw→(get results)
        Axis_velocity_servo→Axis_Command_Output→set_acceleration_command()
        Axis_velocity_servo→Axis_Command_Output→update()
            io2→put(outputvalue);
```

As seen, the Axis module `ax1` method `processServoLoop` performs the basic inputs, computes and outputs expected of a cyclical process. This functionality includes state interpretation so that an Axis module typically has a reference to an Axis FSM. Within the Axis FSM, the calls to `Axis_velocity_servo` are made. For `ax1`, the method `update()` is a wrapper that calls `processServoLoop`.

One assumption among the object interaction is that a state transition, such as `follow_velocity`, is permissible. If not, either the method invocation is ignored or an exception is thrown.

3.4 Proxy Agent Technology

Client/server interaction can be local or distributed. In **local** interaction, the client uses a class definition to declare an object. When a client accesses data or invokes object methods, interaction is via a direct function call to the corresponding server class member. At its simplest, local interaction can be achieved with the server implemented as a class object file or library. Interaction is connected by binding the client object to a newly created server object implementation. Such a binding could be done by static linking, or with a dynamically linked library (DLL) or through a register and bind process that does not use the linker symbol table.

When **distributed** service is needed a *proxy agent* is used which is a set of objects that are used to allow the crossing of address-space or communication domain boundaries[M.S86]. The class describing a proxy agent uses the API of some other class (for which it is a proxy) but provides a transparent mechanism that implements that API while crossing a domain boundary. The proxy agent could use any number of lower level communication mechanisms including a network, shared memory, message queues, or serial lines.

Below is a code example to illustrate the concept of proxy agents. We will assume that we have defined an axis module by the class Axis that has but one method `set_x();`. The following code would be found in the axis module header file (or API specification):

```
class Axis : Environment
{
public:
  void set_x();
```

```
private:
  double myX;
}
```

As a user, one would develop code to connect or bind to the axis module server, which in this case has the name "Axis1." The _bind service is similar to a constructor method, but returns a server reference pointer and keeps track of the number of client pointer references to the server. The bind establishes a client/server relationship with the axis module. The application code is the client, and when Axis methods are invoked, a message is sent to the server. In the following code, the application sets the x variable to 10.0:

```
application(){
    Axis * a1;
    a1 = Axis::_bind("Axis1");
    a1->set_x(10.0);
}
```

If the server is colocated with the application, it is trivial to implement the object server. The `Axis::set_x` implements the value store.

```
Axis::set_x(double _x){ myX = _x; }
```

However, for distributed communication, `Axis::set_x` is defined twice - once on the client side and once on the server side. On the client side we set up the remote communication, which in this case, is a sketch of a remote procedure call.

```
Axis::set_x(double _x){
    callrpc(host, prognum, versnum, procnum, inproc, in, outproc, out)
}
```

On the server side, a server waits for service events (such as the `bind`, and the `set_x` method). A corresponding `Axis::set_x` is defined to handle the x variable store. The server technology could handle events in the background or use explicit event handling. In either case, the server actions are transparent to the client.

```
Axis::set_x(double _x){ myX = _x; }

server(){
    /* register rpc server name */
    while(1) { /* service events */ }
}
```

Given the proxy agent fundamentals, the next step is to adapt this to FSM control. Below is a code sketch of an `Axis` class that defines two methods `process_servo_loop` and `home`. An important aspect of the `Axis` implementation is to make the proxy agent *transparent*. To be transparent, a class must define methods that support local or remote method invocation identically. In order to achieve this, an `FSM` class is defined and when the `home` method is invoked, it inserts a `HOME_EVENT` event into the `Axis` FSM. The FSM has an internal queue for handling events. The FSM may spawn a separate thread of control for event handling.

```
class Axis
{
    FSM AxisFSM;
```

```
    process_servo_loop() { AxisFSM.handle_event(PROCESS_SERVO_LOOP_EVENT); }
    home() { AxisFSM.handle_event(HOME_EVENT); }
};

class FSM {
    msg_queue evq;
    int cur_state;

    handle_event(EV_num)
    {
        evq.send(EV_NO);
     }

    FSM_thread() // optional thread, this could be done in handle_event
    {
        evq.receive(&ev_no);
        call_action(ev_no, cur_state);
    }

    home_update_action() { /* enable homing control plan unit */ }
    process_servo_loop_action() { /* evaluate state */ }
};
```

Within OMAC API, in order to achieve transparency across implementations, all methods contain a parameter field to allow customization of the infrastructure by defining an environment variable at the end of the parameter list. This is an implicit augmentation performed by an IDL compiler. For any OMAC API calling parameter list, the `ENVIRONMENT` parameter appears at the end of the calling sequence, as in:

```
void move(double x, double y, double z, ENVIRONMENT env = default);
```

The `ENVIRONMENT` can be used in several ways to tailor the infrastructure, such as to specify the remote communication protocol and the necessary parameters during transmission. The `ENVIRONMENT` can also be used to set an invocation time-out value or to pass security information. The `ENVIRONMENT` can be a stubbed dummy and ignored by the called method.

The goal of the `ENVIRONMENT` parameter is to provide transparency between invoking function calls locally or invoking function calls remotely. To provide for transparency between local and remote calls, the `ENVIRONMENT` parameter field has a default argument initializer so that local (or remote) calls need not supply this parameter.

The actual infrastructure supported by the `ENVIRONMENT` parameter will not be specified within this OMAC API document. Systems with a proprietary remote communication technology may use the `ENVIRONMENT` parameter field to enable distributed processing. The `ENVIRONMENT` can also be used as a trap door to hide other nonstandard operations. To enable compatibility with known remote processing requirements, OMAC API uses accessor functions to manipulate object data members. The data format creates one or two accessor functions – one to set and one to get – as defined by the cases for read only, write only, or read-write combinations.

```
void set_x(double inx, ENVIRONMENT env=default);
double get_x(ENVIRONMENT env=default);
```

Note that the `ENVIRONMENT` parameter at the end of the parameter list is necessary.

3.5 Infrastructure

The infrastructure deals primarily with the computing environment including platform services, operating system, and programming tools. Platform services include such items as timers, interrupt handlers, and inter-process communications. The operating system (OS) includes the collection of software and hardware services that control the execution of computer programs and provide such services as resource allocation, job control, device input/output, and file management. Real Time Operating System Extensions can be considered platform services since these extensions are required for semaphoring, and pre-emptive priority scheduling, as well as local, distributed, and networked interprocess communication. Programming tools include compilers, linkers, and debuggers.

The OMAC API does not specify an infrastructure because many of the infrastructure issues are outside the controller domain and would be better handled by the domain experts. Further, it is more cost-effective to leverage industry efforts rather than to reinvent these technologies. For example, commercial implementations of proxy agent technology are available. Microsoft has developed and released DCOM (Distributed Common Object Model) for Windows 95 and Windows NT. Many implementations of CORBA (Common Object Request Broker Architecture) are available and Netscape incorporates an Internet Interoperable ORB Protocol (IIOP) inside its browser. The question concerning the hard-real-time capability of such products remains. But, industry is acting to solve this problem. In the interim, control standards that could provide a real-time infrastructure are available [OSA96].

Because there are so many competing infrastructure technologies, OMAC API has chosen to allow the market to decide the course of the infrastructure definition. As such, to achieve plug-and-play module interchangeability, a commitment to a *Platform + Operating System + Compiler + Loader + Infrastructure suite* is necessary for it to be possible to swap object modules.

3.6 Behavior Model

For the OMAC API, *behavior* in the controller is embodied by finite state machines (*FSM*). OMAC API uses state terminology from IEC1131[IEC93]. An FSM *step* represents a situation in which the behavior, with respect to inputs and outputs, follows a set of rules defined by the associated *actions* of the step. A step is either *active* or *inactive*. *Action* is a step a user takes to complete a task which may invoke one or more functions, but need not invoke any. A *transition* represents the *condition* whereby control passes from one or more steps preceding the transition to one or more successor steps. Zero or more actions shall be associated with each step.

3.6.1 Levels of Finite State Machines

For an OMAC API module, there can be nesting of a FSMs. OMAC API does not dictate the levels of FSM. In general, an outer "administrative" FSM exists to handle activities that include initialization, startup, shutdown, and, if relevant, power enabling. The administrative FSM must follow established safety standards. When the administrative FSM is in the `READY` state, it is possible to descend into a lower level FSM. OMAC API defines the OMAC Base Class module to provide a uniform *administrative* state model across modules. The OMAC Base Class state model is illustrated in Figure 10. The administrative state model describes the start-up, shutdown, enabled/ready, configured, aborted, and initialization operations that form the baseline of a module state space. States have methods (e.g., *init()*, `startup()` to cause state transitions.

To enter into a lower FSM, the module enters into the "executing" state as shown Figure 10. In the "executing" state, client/server coordination uses a lower FSM for coordination. This lower FSM is module and application dependent. This lower FSM in turn can have a FSM embedded within it and further nesting of embedded FSM is possible.

Figure 11 shows the nesting of FSM levels. The nesting of one or more lower level *operation* FSM is possible depending on system complexity. Within the nesting of FSM shown in Figure 11, one may have

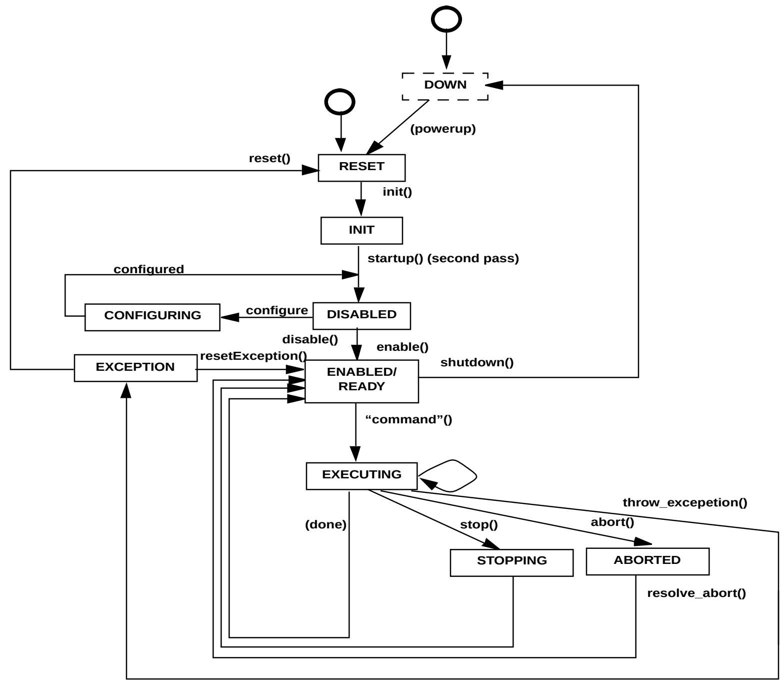

Figure 10: Generalized State Diagram

an "operational" FSM to handle different NC modes corresponding to "auto", "manual", or "MDI". For example, at the operation level for part programming, there may be another level of FSM to handle a family of parts. When a particular part is specified, it may invoke a nested FSM that specifies processing to be performed specific to that part. The designer of a particular control system determines the number of nested FSM levels, depending upon the complexity and organization of the controlled system. At the lowest level FSM is a *dominion* FSM that is the current focus of control.

The levels of FSM fit within OMAC API computational paradigm. Figure 12 shows the OMAC API module general computational paradigm. Within the OMAC API general computational paradigm, an OMAC API module contains a queue, possibly of length 1, for queuing commands. Commands are in the form of FSM. The OMAC API module may have one or more FSM executing on a dominion FSM list. The *dominion* FSM list contains FSM that "rule" over other objects. In the diagram, the FSM are represented by a rectangle within a diamond. The dotted line indicates an optional FSM.

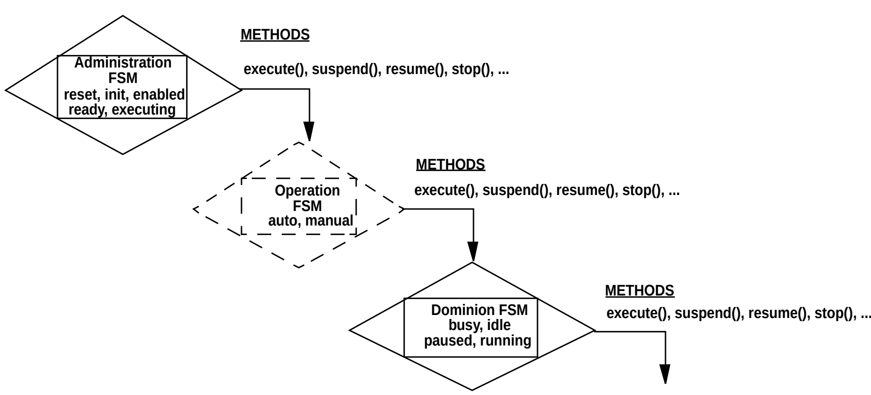

Figure 11: Levels of FSM

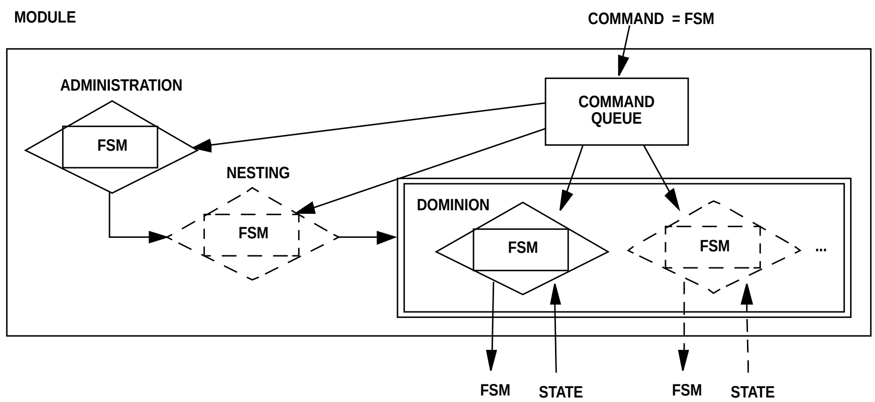

Figure 12: Module Computational Paradigm

With this FSM paradigm, different OMAC API modules have different command queue and FSM dominion list sizes. The Task Coordinator has a one-element queue as well as a one-element dominion FSM. The Discrete Logic module may have a one element queue, but generally has a multi-item dominion FSM list, some active, some not active, to coordinate the IO points. The Axis Group has a minimum two-element command queue, and generally a one-element dominion FSM list unless some blending of operations or synchronization with a spindle FSM is required. The Axis module has an one FSM derived from the OMAC Base Class, but has no command queue. These differences will be further explored.

3.6.2 Control Plan Units

For OMAC API, the FSM is the principal element of both the data flow and control flow. When the FSM is passed between modules it is called a *ControlPlanUnit*. ControlPlanUnits are passed from the sending OMAC API module to the receiving OMAC API module to effect behavior. ControlPlanUnits are then used within a module to handle the control flow. A module executes an ControlPlanUnit until it ends or is superceded by another ControlPlanUnit. How the ControlPlanUnits are implemented is not important to the OMAC API. The following is a sketch of the ControlPlanUnit API.

```
interface ControlPlanUnit
{ // approximate  a graph structure
  ControlPlanUnit execute_unit(); // return next ControlPlanUnit

  void set_active();    // set when "executing"
  void set_inactive();
  boolean isActive();   // for HMI to determine when active

  // ... methods for persistence data in binary or neutral format
  // ... methods for graph representation for navigation purposes
}
```

The CPU is the base class, but the OMAC API defines several specializations. For instance, the HMI ControlPlanUnits for the Task Coordinator called Capabilities. A ControlPlanGenerator, such as one for RS247D or IEC1131, also generate Control Plans for the Task Coordinator, or possibly Control Plans that by-pass a Task Coordinator if one doesn't exist in the architecture. When the TaskCoordinator accepts a ControlPlanUnit from ControlPlanGenerator, the CPU may contain an embedded CPU so that the embedded CPU are then sent to AxisGroup modules or DiscreteLogic modules. The AxisGroup and the DiscreteLogic modules accept a CPU specialization. Figure 13 illustrates the `ControlPlanUnits` hierarchy of possible

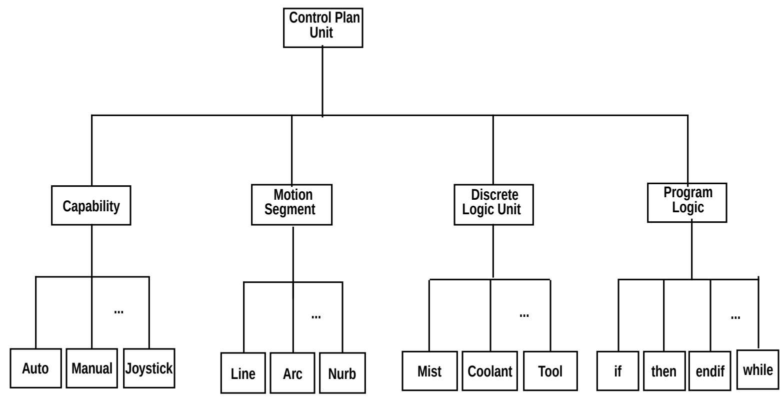

Figure 13: Control Plan Hierarchy

`ControlPlanUnit` specializations. It is through CPU specialization that adding capabilities (such as a NURB MotionSegment) is possible. Specialization of CPU include:

Capabilities corresponds to different machine modes (manual, auto). When the Capability FSM is in the `READY` state, the Capability can descend into a lower FSM or ControlPlanUnit. For example, once in the auto Capability FSM, a lower level FSM for the "cycle" ControlPlanUnit can be used to sequence through a series of ControlPlanUnits.

MotionSegments corresponds to the FSM input for an Axis Group module. In addition to the FSM command and parameter methods, a MotionSegment includes such information as rate, geometry, and a reference to a velocity profile generator that are necessary for trajectory planning.

DiscreteLogicUnits corresponds to the FSM input for an Discrete Logic module. DiscreteLogicUnits coordinate and control an aggregation of IO points. In addition to the FSM command and parameter methods, a DiscreteLogicUnit contains the information necessary for defining asynchronous logic, the event or condition trigger or for defining synchronous logic, the scan rate and FSM.

ProgramLogic CPU for decision making. (e.g., loops, end program and if/then/else).

A `ControlPlanUnit` is responsible for its own branching. For this reason, the method `execute_unit()` return a reference to the next ControlPlanUnit. A `ControlPlanUnit` may embed other `ControlPlanUnits`. A series of `ControlPlanUnit(s)` is a `ControlPlan`. A `ControlPlan` can be a simple list to represent sequential behavior or a complex tree to represent parallel controller behavior. Traversal methods are defined within a ControlPlanUnit so that external modules, such as the HMI, can monitor progress of ControlPlan via the `isActive()` method. Figure 14 illustrates some possible connections of ControlPlanUnits. Through the use of ProgramLogic CPU, one can achieve a mapping from most computer programming control constructs into a list representation.

The `ControlPlanUnit` can contain other ControlPlanUnits. When activated, a CPU can send another CPU to the lower level server. Thus, CPU can be "intelligent" and understand how to coordinate and sequence the lower level logic and motion modules. Consider the following examples. The `ControlPlanUnit` could put `MotionSegments` on the AxisGroup motion queue. The `ControlPlanUnit` FSM can either put `LogicUnits` on the DiscreteLogic queue or activate `LogicUnits` on the DiscreteLogic dominion list similar to a PLC scanning list. Figure 15 illustrates the propagation of CPU through a controller.

To use such a sequence of control, the Control Plan Generator builds a `ControlPlanUnit` for the Task Coordinator FSM that causes a `MotionSegment` FSM to be pushed onto AxisGroup Queue. It is important to understand that this rippling effect is a fundamental mechanism for passing data through an OMAC API controller.

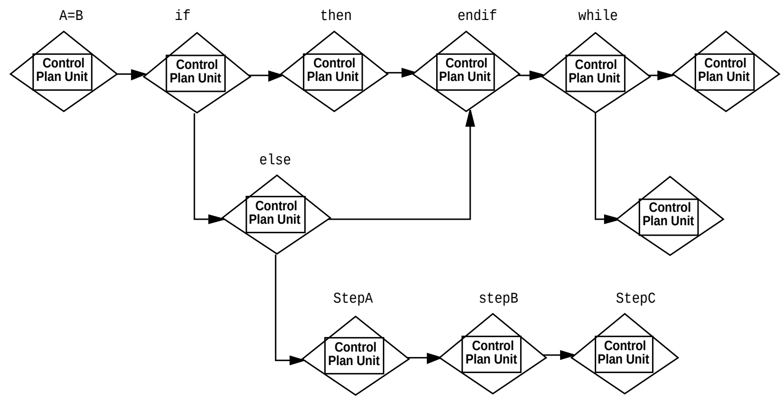

Figure 14: Control Plan

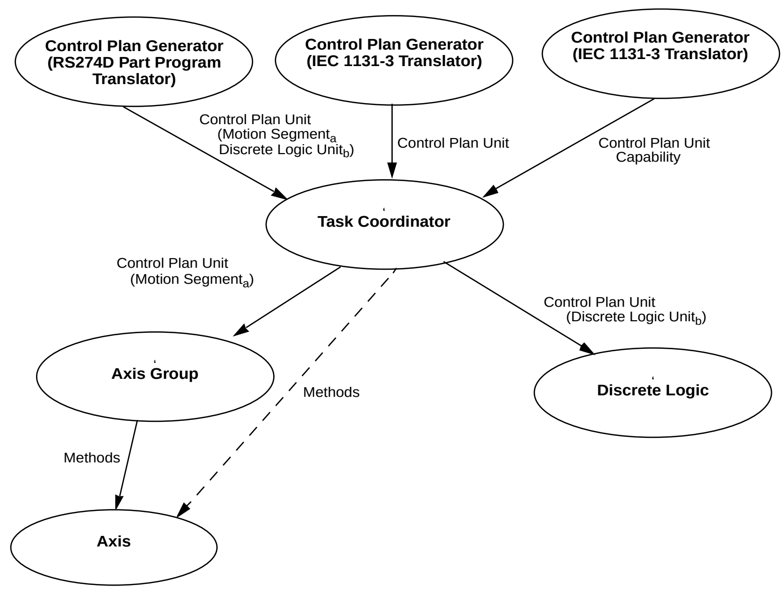

Figure 15: Intelligent CPU Spawning Lower Level CPU

As an example, the application of proxy agent technology will be used by Control Plan Generator to generate a `ControlPlanUnit` for an `axis_homing` FSM. The `axis_homing` is an FSM with a transition method `execute` and a query method `isDone` to determine FSM completion.

```
hmi→taskcoord→set_current_capability(auto);
    taskcoord→auto→start();   //enable this capability
hmi→ControlPlanGen→set_program_name("prog1");
hmi→auto→cycle();                    //start process
taskcoord→update();
    auto→execute_unit();
        cpu=auto→cpg→translate();
        cpu→execute_unit();
            cpu→ag→set_next_motion_segment(ms1);
            cpu→dl→start(dl_toolprep);
```

```
ag→update();
    // read current axes position, overrides, compute fwd kinematics...
    ag→ms1→calc_next_increment(actual_pos, overrides);
    // compute inv kinematics, write axes commanded position
dl→update();
    // scan fsm list (scanning list mechanism hidden from API)
    for(i=0;i<n;i++) dl→scanlist→cpu[i]→execute_unit)();
taskcoord→update();
    auto→execute_unit();
        if(cpu→ag→isDone()) cpu→ag→set_next_motion_segment(ms2);
        auto→cpu→execute_unit();
            auto→cpu→ag→isOK();
            auto→cpu→dl→isOK();
```

The OMAC API specifies that `ControlPlanUnit` objects can embed module references and direct method calls. On the surface this approach appears implausible. However, because of proxy agent technology, creating a "forward reference" by dynamically binding to an object is not hard to do. This dynamic binding is beneficial since it eliminates the need for static encoding of methods with id numbers so that methods can execute across domains (address spaces). To enable forward references, the requirement does exist for the infrastructure to support some "`lookup()`" method to map object names to addresses.

```
interface axis_homing : ControlPlanUnit
{
  attribute MotionSegment ms_homing;  // parameters set by the CPG
    execute()                         // called by Task Coordinator
    {
      if(firsttime)
         ag→set_next_motion_segment(ms_homing); // message passing!
      else if(!ag→isOK());            // do error checking each cycle
    }

  isDone(){ return(!ag→isHomed()); } // called by Task Coordinator
  set_axgrp(char * axgroupname ) { ag=lookup(axgroupname); }
private:
  Axis Group *ag;                     // ag set by the CPG
}
```

The `execute` and `isDone` methods use explicit calls to an Axis Group object. A "forward reference" to the Axis Group object is required. Suppose the Control Plan Generator (`CPG`) receives at constructor time the name "*axisgroup1*" for the Axis Group object. Lookup of the "*axisgroup1*" must be available through the underlying proxy agent technology. Without the proxy agent technology, one has to encode the object `ag` and the methods `ag->home` and `ag->isDone`. This extra programming overhead is hidden by the proxy agent technology.

3.6.3 Task Coordinator

The Task Coordinator module accepts a ControlPlanUnit called a `Capability.`

The Task Coordinator has a one-element FSM dominion list. The dominion FSM list is defined by the `Capability` class definition. Associated with the `Capability` FSM is a `ControlPlan` list.

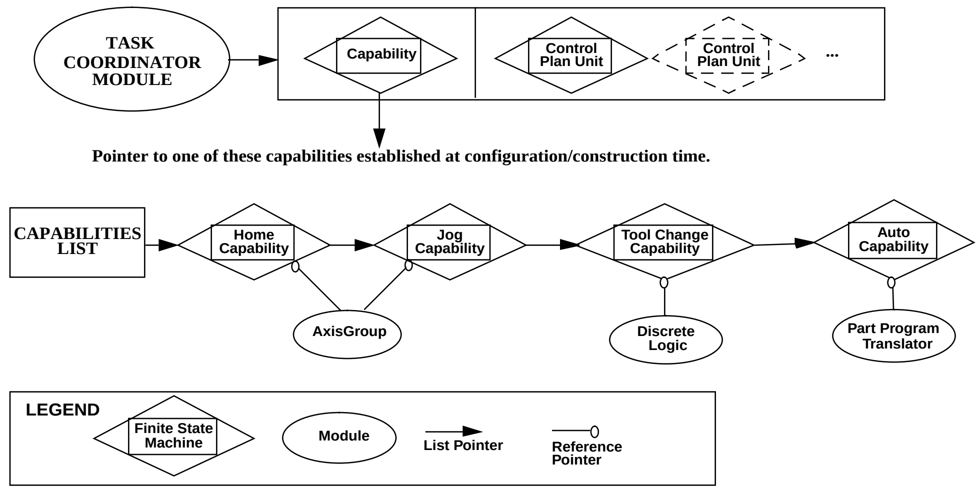

Figure 16: Controller Task Coordinator Capabilities

The `Capability` FSM supports `stop, start, execute, isDone` methods. For an application controller, there is list of capabilities that a Task Coordinator can use. Figure 16 illustrates a typical milling CNC application with `Capability` instances. Each `Capability` has reference pointers to OMAC API modules that it uses. Thus, the `Home Capability` and the `Jog Capability` each have reference pointers to the Axis Group. When a `Capability` is executing, it coordinates the servicing of requests from the HMI. When the `Auto Capability` FSM is executing, it interacts with the Control Plan Generator.

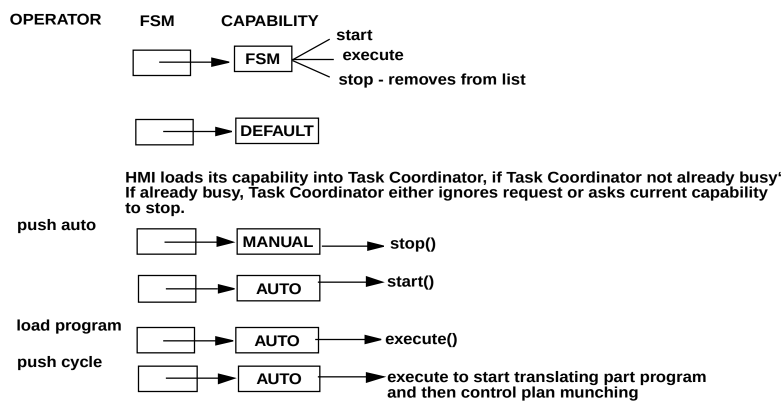

Figure 17: Step Through of a Task Coordinator Capability Sequence

Figure 17 illustrates a sequence of operations that takes a milling CNC from manual mode to automatic mode. The diagram illustrates that a `Capability` FSM has `start, stop, execute` methods. There is the assumption that there is a default `Capability`, probably an `Idle Capability`. In the scenario, the operator pushes the `auto` button that causes the HMI to execute the `Manual Capability stop` method, and load the `Auto Capability` onto the Task Coordinator queue. That cycle, the Task Coordinator will see that the `Manual Capability` boolean `isDone` is True and will swap the `Auto Capability` FSM into the dominion FSM list. The operator action to load a program will result in a program name loaded into the Control Plan Generator. When the operator pushes the cycle button, it will cause the Auto `Capability` FSM to start sequencing Control Plan Generator generated information. Control Plan Generator information is called `ControlPlan` and will covered in the next section.

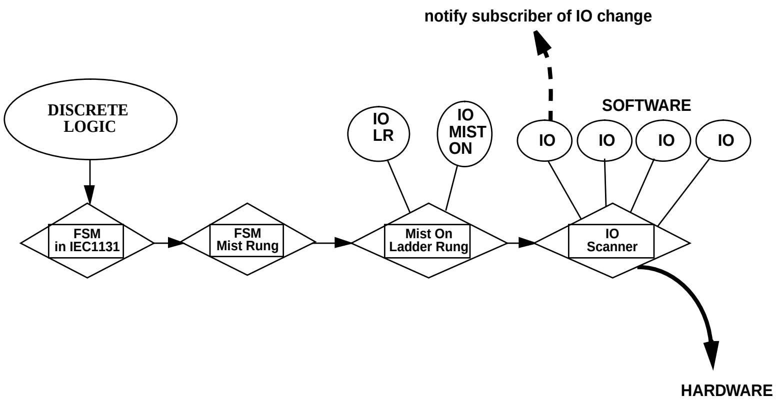

Figure 18: Discrete Logic FSM List

3.6.4 Discrete Logic

The Discrete Logic module is similar to the Task Coordinator module in that it sequences and coordinates actions through dominion FSM. However, for clarity, instead of a monolithic one-element dominion FSM, the Discrete Logic module has a multi-item dominion FSM list. In general, a Discrete Logic dominion FSM could be coded in any of IEC-1131 languages. Figure 18 illustrates the types of FSM that may be found on the Discrete Logic dominion list for a typical milling CNC application. An FSM to handle IO scanning would be expected. An FSM implemented as a Ladder Rung could be expected to handle a relay for turning a Mist pump on. Below one finds a sketch of the activity for turning the IO mist pump on.

```
mist_pump_on_rung()
execute()
{ logic:  trigger relay to turn pump on
          wait till IO/pt says pump is on
          IOmist← on;
}
```

At a higher level, a hardware-independent Mist FSM would be required to coordinate turning Mist on and off. Below is a sketch of pseudo code to sequence the Mist on operation. For coordination between FSM logic, polling or event-drive alternatives exist to wait for the IO Mist on activity to complete.

```
mist_on_fsm()
{ "MistOn LR IO <- on"  to turn LR=ladder rung on
  "subscribe to event that IO Mist On ==on"
  "wait for event or poll for IO point for Mist On == on "
  "done - deactivate FSM for scanning"
}
```

3.7 Foundation Classes and Data Representation

Exchange of information between modules relies on standard information representation. Such control domain information includes units, measures, data structures, geometry, kinematics, as well as the framework component technology. Figure 19 portrays the conceptual organization of framework component software as defined by foundation classes.

Consider the analogy of building materials. The primitive data types, shown at the bottom of Figure 19, are similar to such raw materials as sand, gravel, and clay. Using foundation classes and aggregating

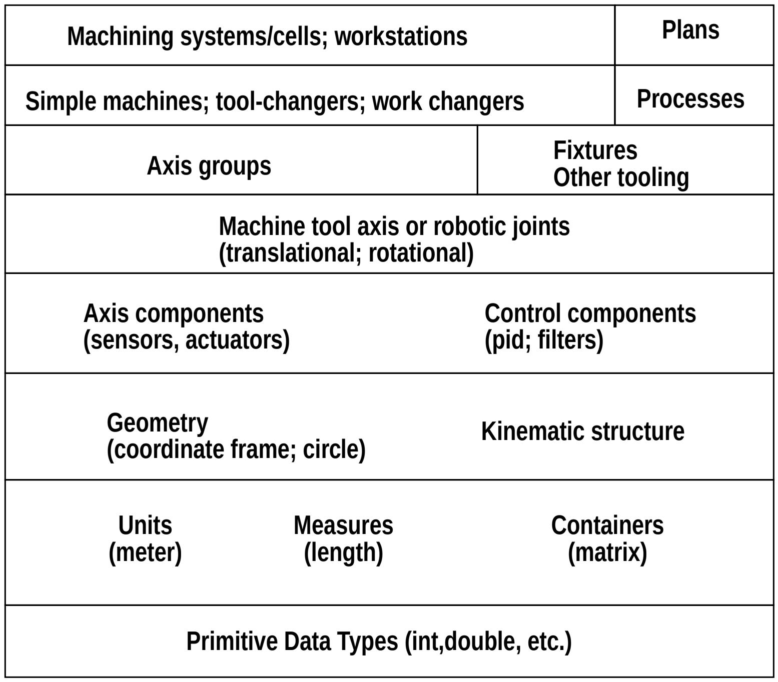

Figure 19: Software Reusable Assets

structural components, a control hierarchy of reusable software components can be built. Based upon the reusable foundation classes, these assets can be used to build class libraries for such motion components as sensors, actuators, and pid control laws.

Not all software objects have physical equivalents. Objects such as axis groups are only logical entities. Axis groups hold the knowledge about the axes whose motion is to be coordinated and how that coordination is to be performed. Services of the appropriate axis group are invoked by user-supplied plans (process programs).

OMAC API has chosen two levels of compliance for data definitions. The first level defines named data types to allow type-checking. The OMAC API uses the IDL primitive data types and builds on these data types to develop the foundation classes and framework components. For control domain data modeling, the OMAC API used data representations found in STEP Part Models for geometry and kinematics [Inta, Intb]. Internally, one could, of course, use any desired representation. The STEP data representations were translated from Express into IDL. Representation units are assumed to be in International System of Units, universally abbreviated SI. Below is the basic set of data types which use STEP terminology for data names but reference other terms for clarification.

Primitive Data

- IDL data types include *constants*, *basic data types* (float, double, unsigned long, short, char, boolean, octet, any), *constructed types* (struct, union and enum), *arrays* and *template types* bounded or unbounded sequence and string.
- IEC 1131 types - 64 bit numbers
- bounded string

Time

Length

- Plane angle

- Translation commonly referred to as position
- Roll Pitch Yaw (RPY) commonly referred to as orientation
- STEP notion of a Transform which is composed of a translation + rpy, also commonly referred to as a "pose."
- Coordinate Frame which is defined as a Homogeneous Matrix

Dynamics

- Linear Velocity, Acceleration, Jerk
- Angular Velocity, Acceleration, Jerk
- Force
- Mass
- Moment
- Moment of Inertia
- Voltage, Current, Resistance

The second level provides for more data semantics. The OMAC API adopted the following strategy to handle data typing, measurement units, and permissible value ranges. Distinct data representations were defined for specific data types. For example, the following types were defined in IDL to handle linear velocity.

```
// Information Model - for illustrative purposes
typedef Magnitude double;

// Declaration
interface LinearVelocity : Units {

    Magnitude  value; // should this value be used?
    // Upperbound and Lowerbound, both zero ignore
    Magnitude ub, lb; // which may be ignored

    disabled();
    enabled();
};

// Application
LinearVelocity vel;
```

In this case, linear velocity is a special class. Unit representation is inherited from a general units model. Permissible values are defined as a range from lowerbound to upperbound. The units and range information are optional and may not be used by the application.

Another data typing problem that must be resolved concerns the use of a parameter. Not all parameters are required or need be set by every algorithm. For example, setting the jerk limit may not be necessary for many control algorithms. To resolve the parametric dependency issue it was decided to use a special value to flag a parameter as "not-in-use". This approach seems simpler than having a `use_xxx` type method for each parameter. For now, OMAC API has decided that setting a parameter to a unrealistic "Not in use Number" (but not actually "Not a Number") value - such as `MAXDOUBLE` or 1.79769313486231570e+308 - renders a `double` parameter to be ignored or not-in-use. A similar number would be required for an integer. This works for level 1 and level 2. Within level 2, the methods `enable` and `disable` were added to explicitly indicate use of a parameter.

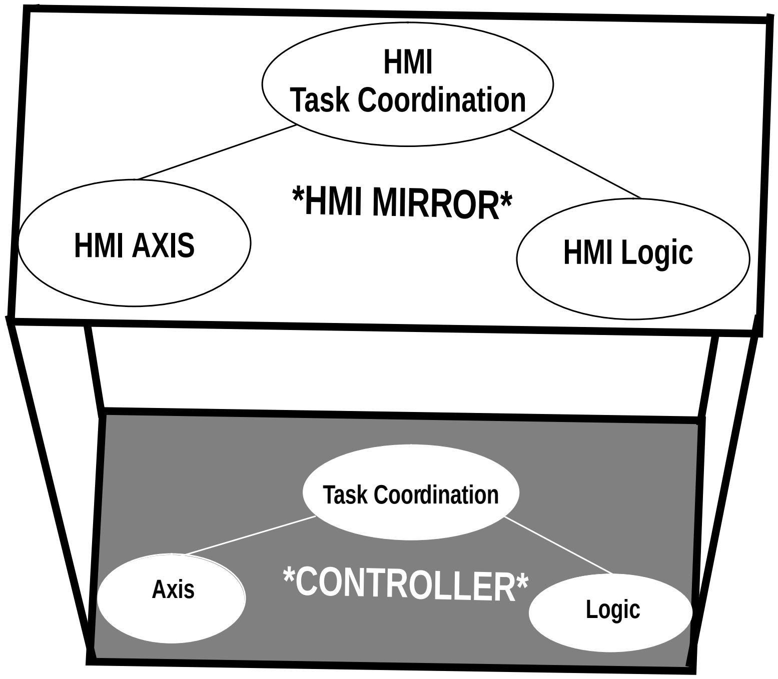

Figure 20: Human Machine Interface Mirrors Controller

3.8 Human Machine Interface

The primary HMI objective of the OMAC API is to provide the ability to "bolt-on" a Human Machine Interface to the controller. The HMI is intended to be independent of the choice of presentation medium, the dialogue mechanism, the operating system, or the programming language.

OMAC API specifies that every controller object has a corresponding HMI object "mirror". A simplifying assumption is that HMI objects communicate to control objects via proxy agents. Figure 20 illustrates the mirroring of a one axis controller that uses a task coordination module for coordination and sequencing in conjunction with a discrete logic module.

The desired HMI functionality is best understood in the context of simple problems. Three "canonical" problems exist that an HMI module must be able to handle regardless of the interface device. First, the user must be able to receive *solicited information reports* about the state of the controller, such as a current axes position. Second, the user must have *command capabilities* such as set manual mode, select axis, and then jog an axis. Third, the user must be alerted when an exception arises, in other words, handle *unsolicited information reports.* Following is an analysis of how the HMI mirror handles these cases.

To handle the information report functionality, an HMI mirror acts as a remote data base that replicates the state and functionality of the controller object and then adds different presentation views of the object. These HMI mirrors are not exact mirrors of the controller state, but rather contain a "snapshot" of the controller state. Figure 21 illustrates the interaction of the HMI mirror and the control object. In the basic scenario of interaction, the control object is the server and the HMI mirror object is the client. Each HMI mirror uses the accessor functions of get and set to interact with the control object. You will notice that each host controller object and corresponding HMI mirror have a proxy agent to mediate communication.

To handle command functionality, the HMI mirror contains the same methods as the controller object so that a command is issued by invoking a method remotely.

To handle abnormal events when polled monitoring may not be possible, an HMI mirror must serve as a client to the control object so that it can post alert events. For such unsolicited information reports, the control object uses an event notification function, `update_current_view`, in which to notify the HMI mirror that an event has occurred. This notification in turn may be propagated to a higher-authority object.

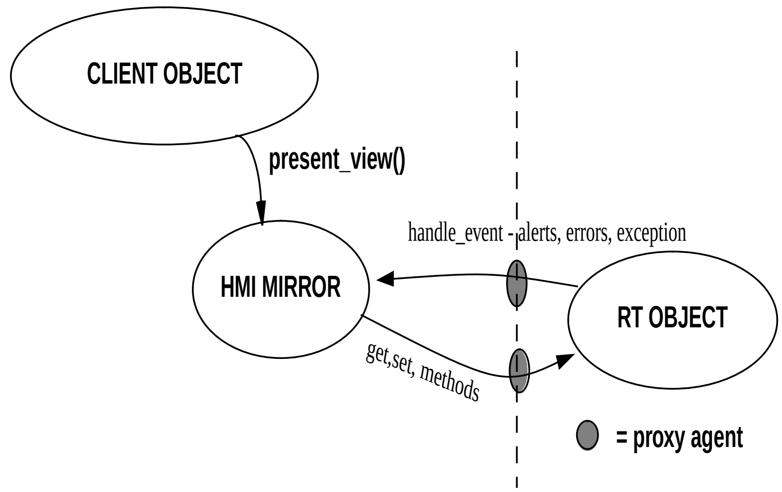

Figure 21: Human Machine Interface

The following HMI definition gives the method extensions that a control object must support to become a mirrored object.

```
interface HMI
{
  // Presentation Methods
  void present_error_view();
  void present_operational_view();
  void present_setup_view();
  void present_maintenance_view();

  // Events - to alert HMI that something has happened
  void update_current_view();
};
```

A benefit to using the HMI mirrors is the potential for vendors to supply a control object, as well as a presentation HMI object that can be incorporated into their Operator Interface. As an example of this technology, a tuning package can provide a Windows-based GUI to do some knob turning. Another example, is a tuning package that offers this capability to be plugged inside a Web browser. With this development, unlimited component-based opportunities are available.

4 API

Technical Note: These API are for review and comment only. There is no guarantee of correctness. This specification approximates the intended direction of the final API.

4.1 Disclaimer

This software was produced in part by agencies of the U.S. government, and by statute is not subject to copyright in the United States. Recipients of this software assume all responsibility associated with its operation, modification, maintenance, and subsequent redistribution.

As of April 30, 1997 the IDL Definitions are in state of transition. Some "pure" IDL some hybrid. Eventually there will be no attributes, only get and set methods on these attributes, since IDL does not produce a get/set prefix to the methods. This will not work for non-CORBA-like systems.

4.2 Basic Types

```
// All class definitions should register with central name/type server
interface OMAC_CLASS
{
  attribute char * name;
  attribute char * type;

};

interface OMAC_MODULE
{
  // Administrative State Transition Methods
  void  estop();
  void  reset();
  void  init();
  void  startup();
  void  enable();
  void  disable();
  void  execute();
  void  shutdown();

  void  throw_exception();
  void  resolve_exception();

>  void  stop();
>  void  abort();


  boolean isReset();
  boolean isInited();
  boolean isEnabled();
  boolean isDisabled();
  boolean isReady();
  boolean isEstopped();
>  boolean isException();

};

// Level 1 - these will be backed out from the other API definitions
//
typedef long                API;
typedef double              AngularVelocity;
typedef Boolean             boolean;
```

```
typedef Translation         CartesianPoint;
interface                   CoordinateFrame { /* FIXME */ } ;
typedef double              Force;
typedef double              Length;
typedef double              LinearVelocity;
typedef double              LinearAcceleration;
typedef double              LinearJerk;
typedef double              LinearStiffness;
interface                   LowerKinematicModel {/* FIXME */ } ;
interface                   MaintHistory  { /*FIXME*/ };
typedef double              Magnatude;
typedef double              Mass;
typedef double              Measure;
enum                        MOT_OBJ { SPEED, ACCURACY};
typedef double              PlaneAngle;
interface                   RESOURCE  {/*FIXME*/ } ;
interface                   RPY  {/*FIXME*/ } ;
interface                   RWCollectable{/*Something equivalent to RogueWave */};
typedef                     RWOrdered{/*Something equivalent to RogueWave */};
interface                   Time { /* FIXME */ };
interface                   Translation  {/*FIXME*/ } ;
interface                   UNITS {/*FIXME*/ } ;
interface                   UpperKinematicModel {/*FIXME*/ } ;


//?? Or you can assume numbers are flagged not active at
//?? construction time.
// Below most control parameters would be typed as double
#define double_not_active 1.79769313486231570e+308
#define long_not_active 0x80000000
#define short_not_active 0x8000


// Level 2  Example - not defined here

interface LinearVelocity : Units  {
    Magnitude  value; // should this value be used?
    // Upperbound and Lowerbound, both zero ignore
    Magnitude ub, lb; // which may be ignored
    disabled();
    enabled();
};
interface Units{ /* FIXME */ };
```

4.3 Control Plan

```
interface ControlPlanUnit
{ // approximate  a graph structure
  ControlPlanUnit execute_unit(); // return next ControlPlanUnit
  // ControlPlanUnit get_next_unit();

  void set_active();    // set when "executing"
  void set_inactive();
  boolean isActive();   // for HMI to determine when active

  // persistence data a la binary image
  void save(char * file);
  void restore(char * file);

  // persistence data in neutral format (pre-configuration)
  void save_neutral(char * file);
  void restore_neutral(char * file);

  // The graph is used for non-execution navigatation
  ControlPlanUnit * cpu[100 /*max*/];
  attribute int length;  // number of arcs in this graph node
  attribute int max;     // max number of arc possible should grow dynamically
  // FIXME: add traversal functions here
};

interface ControlPlan : RWList<ControlPlanUnit>  {};
```

4.4 Scheduling Updater

```
interface updatable
{
  attribute double period;
  void update()=0;
]

interface asych_updater
{
  register_updatable(updatable upd);
  virtual void update();

}

interface periodic_update : asynch_update
{
   get_timing_interval();
   virtual void update();
```

```
}
```

4.5 IO

```
// Level 1
interface IO_PT<T>   // <T>= int, boolean, double, float
{
  <T> get_value();
  void set_value(<T> v);

  char *get_name();
  void *set_name(char *name);


  attribute (void *) (*monitor) ();  // is an independent thread of execution

  //?? attribute device-info;   // reference to device info

  //?? attribute int type;   // 1=read-only, 2=read/write, 0=don't care
  // or use IO derived type to differentiate types
  //?? attribute UNITS units;
  //?? attribute <T> upper_bound;
  //?? attribute <T> lower_bound;
};

interface callbackNotification
{
  void execute();
};

interface IOPt_Notify
{
  void notify_handlers(); /* list management */
  void attach(callbackNotification cb);
};

// example derived type
// interface IOPt_Nofity_on_sign_change: IOPt_Notify { } ;

typedef sequence<int> IOvalues;
typedef sequence<char *> IOnames;
typedef sequence<char *> IOmetadata;

// Or should this just be an array of IOPts?
interface IOgroup
{
```

```
  IOvalues get_values();
  void set_values(IOvalues);

  void add_IO_PT(IO_PT<T> io);
  IOnames get_names();
  IOmetadata get_metadata();

};

//??typedef sequence<IO_PT<T>> IO_SYSTEM;
// A container for a list of IO Points
interface IOsystem
{
  IOgroup get_IO_GROUP(char * name);
  IOPt get_IO_PT(char * name);
};


// Example
interface myIO : IO_SYSTEM /*UPDATABLE*/
{
  IO_PTshort encoder1;
  IO_PTshort encoder2;
  IO_PTlong encoder3;

  void update();
  callbackNotification new_sample_available; /* tell clients of new data */
  set_pacer_clock(divisor);  /* control */
};

// Level 2: Hierarchy of Common IO Points - for type checking
// See IO API Document for further details
```

4.6 Task Coordinator

```
// Each capablity is an FSM and types of capabilities include: manual, auto, estop, etc.
// FIXME: What is the relationship of manual to auto and any to estop?
// Internally the capbility is a FSM.
interface Capability
{
  void start();
  void execute();
  void update_cap(); //update() can call update_cap()
  void stop();
  void abort();
  void throw_execption();
  void resolve_execption();
  boolean isDone();
```

```
  boolean isActive();   //+
};

typedef sequence<Capability> Capabilities;

// Task Coordinator accepts one capability from a list of capabilities.
interface TaskCoordinator : OMAC_MODULE /*UPDATABLE*/
{

  virtual void update();  //can be inherited from UPDATER

  // Capability List Management
    void add_to_list(Capability * cap);
    void remove_from_list(Capability * cap);
    Capabilities get_list();

  // Current Capability Management
    Capability * get_current_capability();
    void set_current_capability(Capability *  cap);
};

```

4.7 Discrete Logic

```
// Discrete Logic Module contains a list of logic units. A PLC like scan
// goes down the list and executes each logic unit if it is on. Logic units
// will be executed as often as its posted scan rate indicates.
// Internally each discrete logic unit is an FSM.
// Discrete Logic Units (DLUs) are grouped by scan rates.
interface DiscreteLogic : OMAC_MODULE
{

  // Logic Units Management
    DiscreteLogicUnit * create_discrete_logic_unit();
    void add_logic_unit(DiscreteLogicUnit dlu);
    void remove_logic_unit(DiscreteLogicUnit dlu);
    void enable_logic_unit(DiscreteLogicUnit dlu);
    void disable_logic_unit(DiscreteLogicUnit dlu);
};

// Derived from ControlPlanUnit, see: part program translator
interface DiscreteLogicUnit: ControlPlanUnit
{
  attribute integer interval;

  void start();
  void scan_update();
  void stop();
```

```
  boolean isOn();
  boolean turnOn();    // external event causes invokes this method
  boolean turnOff();
};

```

4.8 Control Plan Generator

```
// Level 1 assuming simple File Manipulation
interface ControlPlanGenerator
{
    void set_program_name(char * s);
    char * get_program_name(char * s);

    boolean  checkSyntax();
    char * get_error_codes(); // or returns file name or file pointer?

    ControlPlan  translate(); // complete translation into ControlPlan
    ControlPlanUnit * get_next_ControlPlanUnit();  // step by step translation
};

// Level 2  Production Data Management
interface ProductionDataManagement : FILE VERSION
{
  // A standard should be completed by 9/97
};
interface CPGLevel2
{
    attribute ProductionDataManagement pdm;
};

// Defer interface specification to CAD

```

4.9 Axis Group

There are some inconsistencies within the Axis Group module API. The major remaining problem is to resolve the use of the axis group velocity profile generator (VPG) versus having the VGP embedded within a motion segment.

```
//+ add accel mode - use instead of enum - windows problem
typedef int ACC_MODE;
#define  S_CURVE 1
#define  TRAPEZOIDAL 2

interface AxisGroup : OMAC_MODULE UPDATABLE
{
//+  enum { ERROR, HELD, HOLDING, STOPPED, STOPPING, PAUSED, PAUSING, RESUME, EXECUTING, IDLE };
```

```

    // STATE LOGIC
    // ================================================

    void hard_stop_axes();  // Stop at max deceleration rate (abort)
    void pause_axes();      // stop on path
    void hold_axes();       // stop at end of segment
    void resume_axes();     // Resumes motion from current point

  //  void     update_axes();
    void    update();       //+ changed for consistent interface

    int get_current_state();
    String get_current_state_name();
    boolean is_OK();
    boolean is_Executing();
    boolean is_Held();
    boolean is_Holding();
    boolean is_Paused();
    boolean is_Pausing();
    boolean is_Stopping();
    boolean is_Stopped();

    // These methods could be operator Control Plan Unit
    void jog_axis( int axis_no, VelocityMeasure speed );
    void home_axis( int axis_no, VelocityMeasure speed );
    void move_axis_to(int axis_no, VelocityMeasure speed,  LengthMeasure  to_position);
    void increment_axis(int axis_no, VelocityMeasure speed, LengthMeasure increment);

    // BUFFERING MANAGEMENT
    //================================================
    void set_next_motion_segment( Motion_Segment block);
    // Motion_Segment get_current_motion_block( );  //hazardous to your controller's health
    int get_MaxqSize() const;    // largest queue size possible=n
    void set_qLength(int value); // maximum number of queue members=(1..n)
    int get_qLength() const;
    int get_current_qSize();     // number of items in queue=i
    boolean is_Full();           // number of items = n
    boolean is_Empty();          // number or items = 0

    void flush();                // flush all segments
    void skip();                 // skip to next segment
    void save_q_context();       // save current queue
    void restore_q_context();    // restore saved queue

    // FIXME: possibly more queue mgt functions (accessor, query, ... )

    // CONVENIENCE FUNCTIONS TO ACCESS MOTION SEGMENT DATA
```

```
//=================================================
length_measure * get_neighborhood() const;
LinearVelocity * get_feedrate() const;
VelocityMeasure * get_traverseRate();
double get_feedrateOverride() const;
double get_spindlerateOverride() const;
LinearJerk * get_jerkLimit() const;
Boolean get_inPosition() const;
void set_inPosition(Boolean value); /* private method*/

// See Note 1
Measure get_actual_axis_position( int axis_no );
OacVector get_actual_axes_positions( );
CoordinateFrame get_xformed_actual_positions();
Measure get_commanded_axis_position( int axis_no );
OacVector get_commanded_axes_positions( );
CoordinateFrame get_xformed_commanded_positions( OacVector axis_positions);

ACC_MODE get_accMode() const;

// KINEMATIC INFORMATION
//=================================================
// Axis under control
CoordinatedAxes * get_coordinatedAxes() const;
KinStructure * get_kinStructure() const;
void set_kinStructure(KinStructure * value);
ToolPartTransforms * get_toolTransform() const;
CoordinateFrame * get_baseFrame();
void set_baseFrame(CoordinateFrame * value);

// recovery from fault error, sharing
void    inhibit_axis( int axis_no, boolean inhibit );
boolean    axis_inhibitd( int axis_no );
void    inhibit_spindle( boolean inhibit );
boolean    spindle_inhibitd();

// TRAJECTORY INFORMATION
//=================================================
void set_blending(boolean flag);      // TRUE=ON, FALSE=OFF
void set_single_step(boolean flag);   // TRUE=ON, FALSE=OFF

// void set_VPG(VelocityProfileGenerator vpg);
// VelocityProfileGenerator get_VPG();

// Timing is now a reference to another object
// time_measure get_axisUpdateInterval() const;
// void set_axisUpdateInterval(time_measure value);
attribute Timing timing;
```

```
    void set_physical_limits(Rate * limits); //+ 3-Jun-1997
    Rate * get_physical_limits();            //+
  };

  // NOTES
  // 1. There is a problem in JAVA with returning data type.
  // Storing into calling parameter as a side effect Side
  // instead of
  //        OacVector get_commanded_axes_positions( );
  // use
  //        void get_commanded_axes_positions( OacVector positions );
  // It is possible to redo above in this signature style.
  // 2. Issue: There are issues as to maximum acceleration of device
  // versus Control Plan Unit (Motion Segment)

  // Control Plan Class Definitions- Motion Segments
  #if 0
  interface CoordinatedAxes
  {
    // Fixme
  };

  interface OacVecetor
  {
    // how does this differ from PathNode
  };

  interface PathNode
  {
    transform get_controltransform();
    void set_controltransform(transform value);
  };
  interface PathElement : public KinematicPath
  {
    virtual void initAccDecProfile(LinearVelocity *vel);
    void set_start_point( PathNode start_point ); // axgroup sets
    PathNode get_start_point( );
    PathNode get_end_point();                     // axgroup sets
    // void set_end_point(PathNode end_point);     // ppt or internal use
    virtual LengthMeasure get_distance_to_go();
    boolean isPathComplete();
    virtual LengthMeasure pathLength();
    // virtual LengthMeasure pathLength(XYZ xyz); // what is this
  };
  #endif
  interface Rate
  {
```

```
  void set_nominal_feedrate(double vnom);
  int set_current_feedrate(double vmax);      // includes override
  int set_maximum_acceleration(double amax);
  int set_maximum_jerk(double jmax);

  double get_nominal_feedrate();
  double get_current_feedrate();              // includes override
  double get_maximum_acceleration();
  double get_maximum_jerk();

  double get_current_velocity();
  void set_current_velocity(double vcur);

  double get_final_velocity();
  void set_final_velocity(double vcur);

  double get_current_acceleration();
  void set_current_acceleration(double acur);

  int get_acc_state();
  void set_acc_state(int val);
  int isDone();
  int isAccel();
  int isConst();
  int isDecel();

  void set_nominal_spindle_speed(double spd); // why here?
  double get_nominal_spindle_speed();
};
interface KinematicInfo
{
  void setToolCenter(LengthMeasure& effectiveDisplacement,
             CRCMODE cutterRadiusCompensation);

  Xform get_current_frame();
  void set_current_frame( Xform current_frame );

  KinematicsModule get_kinematics();
  set_kinematics(KinematicsModule kin);
};

interface VelocityProfileGenerator
{
  AccDecProfile * get_accDecProfile();
  void set_accDecProfile(AccDecProfile * value);

  void set_blending_point_distance( double distance );
  double get_blending_point_distance();
```

```
    time_measure * get_sampling_time();
    void set_sampling_time(time_measure * value);
    /* New  3-Jun-1997 */
    void hold_segment();
    void pause_segment();
    void resume_segment();
  };
  // Base Class for Motion Segment
  // Derived from ControlPlanUnit  - see part program translator
  interface Motion_Segment  : ControlPlanUnit
  {
    attribute KinematicInfo kin;

    void set_vpg(VelocityProfileGenerator * _vpg);
    VelocityProfileGenerator * get_vpg();

    void set_translational_rate(Rate * rate);
    Rate * get_translational_rate();

    void set_orientation_rate(Rate * rate);
    Rate * get_orientation_rate();

    void set_angular_rate(Rate * rate); // does this belong in axis group?
    Rate * get_angular_rate();

    // if internal velocity profile generation supply this interface
    void set_blending_point_distance( double distance );
    double get_blending_point_distance();

    LengthMeasure calc_distance_remaining(); // axes

    void OacVector get_incremental_distance( );
    OacVector get_lengths_remaining();  // per axis
    OacVector calc_next_increment(double feed_override,
                  double spindle_override,
                  //? doesn't this need in current_position
                  double[]    increment = NULL /* ignore side effect */
                  );
    boolean start_next_segment(); //? what does this mean init?
  //?  int init(double cycle_time); //+ 3-Jun-1997
    void pause_segment();
    void hold_segment();  /* new */
    void stop_segment();  /* new 3-Jun-1997 set motion to done */
    void resume_segment();
    boolean is_paused();
    boolean is_held();

```

```
  // Program information (file, line number, block) and signals(active)
  void set_ppb( PartProgramBlock ppb );
  void segment_started();
  void segment_finished();
};
//NOTES:
// 1. Handling Termination Condition:
// a. Exact Stop = blending distance=0

#endif

```

4.10 Axis

```
interface Axis;
interface Axis_Absolute_Pos;
interface Axis_Acceleration_Servo;
interface Axis_Commanded_Output;
interface Axis_Dyn;
interface Axis_Error_And_Enable;
interface Axis_Force_Servo;
interface Axis_Homing;
interface Axis_Increment_Pos;
interface Axis_Kinematics;
interface Axis_Jogging;
interface Axis_Limits;
interface Axis_Maint;
interface Axis_Operation;
interface Axis_Positioning_Servo;
interface Axis_Rates;
interface Axis_Sensed_State;
interface Axis_Setup;
interface Axis_Velocity_Servo;

typedef double Axis_Accel_Cmd;
typedef double Axis_Force_Cmd;
typedef double Axis_Position_Cmd;
typedef double Axis_Velocity_Cmd;

interface Axis : OMAC_MODULE
{
  // Get Reference Objects
  // Axis_Absolute_Pos  get_absolute_position(); // removed 23-Jun-1997
  Axis_Acceleration_Servo get_acceleration_servo();
  Axis_Commanded_Output get_command_output();
  Axis_Error_And_Enable get_error_and_enable();
  Axis_Force_Servo  get_force_servo();
  Axis_Homing get_homing();
```

```
    Axis_Increment_Pos get_increment_position();
    Axis_Jogging get_jogging();
    Axis_Positioning_Servo get_positioning_servo();
    Axis_Sensed_State get_sensed_state();
    Axis_Velocity_Servo get_velocity_servo();

    void set_acceleration_servo(Axis_Acceleration_Servo);
    void set_command_output(Axis_Commanded_Output);
    void set_error_and_enable(Axis_Error_And_Enable);
    void set_force_servo(Axis_Force_Servo);
    void set_homing(Axis_Homing);
    void set_increment_position(Axis_Increment_Pos);
    void set_jogging(Axis_Jogging);
    void set_positioning_servo(Axis_Positioning_Servo);
    void set_sensed_state(Axis_Sensed_State );
    void set_velocity_servo(Axis_Velocity_Servo);

    long processServoLoop( ); // the primary function.
    long check_preconditions( ); //  checked at every servo loop.

    // State transition methods and state queries

    void disable_axis();            // DISABLE_EVENT
    void enable_axis();             // ENABLE_EVENT
    void e_stop();                  // E_STOP_EVENT
    void follow_command_position(); // FOLLOW_POSITION_EVENT
    void follow_command_torque();   // FOLLOW_TORQUE_EVENT
    void follow_command_velocity(); // FOLLOW_VELOCITY_EVENT
    void follow_command_force();    // FOLLOW_FORCE_EVENT
    void home(double velocity);     // START_HOME_EVENT
    void jog(double velocity);      // START_JOG_EVENT
    void reset_axis();              // RESET_EVENT
    void stop_motion();             // CANCEL_EVENT
    void update_axis();             // UPDATE_EVENT

    // Returns a ASCII readable string
    String current_state_name();

    // Instead of:
    // int current_state();
    //   DISABLED                 =  1,
    //   ENABLED                  =  2,
    //   E_STOPPED                =  3,
    //   FOLLOWING_POSITION       =  4,
    //   FOLLOWING_TORQUE         =  5,
    //   FOLLOWING_VELOCITY       =  6,
    //   HOMING                   =  7,
    //   JOGGING                  =  8,
```

```
  //  STOPPING                 =  9;

  // Use accessor functions so there is no confusion about numbering
  // Also inherit state queries from OMAC Base Module
  boolean is_following_acceleration();
  boolean is_following_force();
  boolean is_following_position();
  boolean is_following_velocity();
  boolean is_homing();
  boolean is_incrementing_position();
  boolean is_jogging();
  boolean is_movingto();
};
interface Axis_Acceleration_Servo
{
  // All invoked by Axis FSM
  boolean acceleration_servo_error();
  void acceleration_error_action();
  void acceleration_update_action();
  void end_acceleration_following_action();
  void start_acceleration_following_action();
};
interface Axis_Commanded_Output
{
  Axis_Position_Cmd get_position_command();
  Axis_Velocity_Cmd get_velocity_command();
  Axis_Accel_Cmd get_acceleration_command();
  Axis_Force_Cmd get_force_command();

  void set_position_command( Axis_Position_Cmd  positioning_cmd );
  void set_velocity_command( Axis_Velocity_Cmd  velocity_cmd );
  void set_acceleration_command( Axis_Accel_Cmd acceleration_cmd );
  void set_force_command( Axis_Force_Cmd   force_cmd );

  void update_commanded_output();  // updates using connections to IO

};
interface Axis_Dyn
{
  attribute Force staticFriction;
  attribute Force runFriction;
  attribute Time timeConstant;
  attribute Length backlash;
  attribute Length deadband;
  attribute Mass axmass;

  attribute LinearAcceleration accelerationLimit;
  attribute LinearAcceleration decelerationLimit;
```

```
  attribute LinearJerk jerkLimit;
  attribute LinearAcceleration zeroVelAccLim;
  attribute LinearAcceleration maxVelAccLim;

  attribute Length overshootStepInput;
  attribute Time risingTimeStepInput;
  attribute Force quasiStaticLoadLimit;
  attribute LinearStiffness loadedCaseSpringRate;
  attribute LinearStiffness worstCaseSpringRate;
  attribute Mass inertia;
  attribute Force damping;

};
interface Axis_Error_And_Enable
{
  void reset_axis_action();
  void disable_axis_action();
  void enable_axis_action();
  void e_stop_axis_action();
};
interface Axis_Force_Servo
{
  // All invoked by Axis FSM
  boolean force_servo_error();
  void force_error_action();
  void force_update_action();
  void end_force_following_action();
  void start_force_following_action();
};

interface Axis_Homing
{
  void start_homing_action(double start_velocity ); // prepares homing
  void homing_update_action();          // called each servo cycle
  void stop_homing_action();            // stops homing before completion
  void homing_complete_action();        // On transition from homing to Enabled
                                        // - when homing is completed
  void e_stop_homing_action();          // On transition from homing to E-stopped
  void    disable_homing_action();      // On transition from homing to disabled
  boolean is_done();                    // signals when homing is completed
  boolean is_stopping();
  boolean homing_error();               // true if error has occurred during homing
};
interface Axis_Jogging
{
  void start_jogging_action( double target_velocity );
  void jogging_update_action();
  void stop_jogging_action();
```

```
  boolean is_done();
  boolean is_stopping();
  boolean jogging_error();
  void e_stop_homing_action();
  void disable_homing_action();
};
interface Axis_Kinematics
//    Provision for lower kinematic model and upper kinematic
//    model consistent with ISO STEP standard.

//    Include services for characterizing these errors :

//    Include provision for
//    - geometric errors of motion
//    - thermally induced errors

//    The posFeedBackGain and the velFeedBackGain are
//    calculated using the connectivity of the jointCompts.

//    The basic synthesis model is the ISO standard for
//    kinematic modeling, which is close to the D-H model
//    which had  its genesis in robotics, priimarily oriented
//    toward a single robotic device.  Since manufacturing
//    equipment could consist of multiple such devices working
//    on a single workpiece or a set of workpieces, we extend
//    the ISO kinematic model, to provide for the inclusion of
//    kinematic models for fixtures, workpieces, and tooling.
//    The D-H model is also extended to include kinematic
//    errors of motion, the composed property of interest is
//    the motion of the work-point as a result of motions of
//    the Axis (or vice versa).  The kinematics model also
//    supports the model of dynamics and states.
{
  attribute double Ks;
  attribute double posFeedBackGain;
  attribute double velFeedBackGain;
  attribute UpperKinematicModel ukm;
  attribute LowerKinematicModel lkm;
  attribute CoordinateFrame placement;
};
interface Axis_Limits
//Limits to Motions Ranges
{
  // Misc. parameters
  attribute LinearVelocity maxVelocity;
  attribute LinearJerk JerkLimit;
  attribute Force maxForceLimit;

```

```
    attribute Length usefulTravel;
    attribute Length cutOffPosition;

    // Following Error levels: warning, limit, violation
    attribute Length warnLevelFollError;
    attribute Length followingErrorViolationLim;
    attribute Length followingErrorWarnLim;
    attribute Length followingErrorWarnAmt;

    // Overshoot Error Levels: warning, limit, violation
    attribute Length overshootWarnLevelLimit;
    attribute Length overshootLimit;
    attribute Length overshootViolationLim;
    // Amount of overshoot
    attribute Length overshootWarnLevelAmt;

    // Underreach Error Levels: warning, limit, violation
    attribute Length underreachWarnLevelLimit;
    attribute Length underreachLimit;
    attribute Length underreachViolationLim;
    // Amount of undershoot
    attribute Length underreachtWarnLevelAmt;

    // OverTravel Limits
    attribute Length softFwdOTravelLim;
    attribute Length softRevOTravelLim;
    attribute Length hardFwdOTravelLim;
    attribute Length hardRevOTravelLim;
  };

  interface Axis_Maint
  //     Provision for data and operations that support
  //     maintenance, e.g. health-tests, health-monitoring.
  {
    // Originally * pointer
    attribute MaintHistory mh;
  };
  interface Axis_Positioning_Servo
  {
    // All invoked by Axis FSM
    boolean positioning_servo_error();
    void positioning_error_action();
    void positioning_update_action();
    void end_positioning_following_action();
    void start_positioning_following_action();
  };
  interface Axis_Rates
  {
```

```
  //Specifications of travel capabilities.
  //worst-case conditions.  But to take advantage of more
  //capability provide a model that describes conditions
  //when more capability is available and the corresponding
  //values or value-functions.

  attribute Length maxTravel;
  attribute LinearVelocity maxVelocity;
  attribute LinearAcceleration maxAcceleration;
  attribute LinearJerk maxjerk;
  attribute Length posErrRatioIdleStationary;
  attribute Length posErrRatioIdleMoving;
  attribute Length posErrRatioCutStationary;
  attribute Length posErrRatioCutMoving;
  attribute long repeatability;
};
interface Axis_Sensed_State
{

  //if(!hardFwdOTravel) && if(!softFwdOTravel) &&if(!hardRevOTravel) &&
  //    if(!softRevOTravel)
  //then enablingPrecondition = 1;
  //else enablingPrecondition = 0;
  //    Concurrency: Sequential
  Boolean getEnablingPrecondition();
  void setEnablingPrecondition();

  attribute Boolean inPosition;
  attribute Boolean softFwdOTravel;
  attribute Boolean hardFwdOTravel;
  attribute Boolean softRevOTravel;
  attribute Boolean hardRevOTravel;
  attribute Boolean followingErrorWarn;
  attribute Boolean followingErrorViolation;
  attribute Boolean overShootViolation;
  attribute Boolean enablingPrecondition;

  // Addition 12/12/96
  Axis_Position get_actual_position();
  Axis_Velocity get_actual_velocity();
  Axis_Accel  get_actual_acceleration();
  Axis_Force  get_actual_force();
};
interface Axis_Setup
//Services perparatory to automatic cyclic operation. Data that can be supplied
// before arrival of current motion command.
{
  // FIXME: 23-Jun-1997 Sort out
```

```
  // sets the reference to the axis rates for physical limits, software limits.
  attribute AxRates physicalLimits;
  attribute AxRates currentRates;
  attribute AxDyn AxD;
  attribute MOT_OBJ motionObjective;
};
interface Axis_Velocity_Servo
{
  // All invoked by Axis FSM
  boolean velocity_servo_error();
  void velocity_error_action();
  void velocity_update_action();
  void end_velocity_following_action();
  void start_velocity_following_action();
};
```

4.11 Control Law

```
interface CONTROL_LAW
{
  // Parameters
  void set_commanded(double setpoint);
  double get_commanded();

  void set_commanded_dot(double setpointdot);
  double get_commanded_dot();

  void set_commanded_dot_dot(double setpointdotdot);
  double get_commanded_dot_dot();

  void set_output(double value);
  double get_output();

  void set_feedback(double actual);
  double get_feedback();

  void set_following_error(double epsilon);
  double get_following_error();

  // Offsets
  void set_following_error_offset(double preoffset);
  double get_following_error_offset();

  void set_output_offset(double postoffset);
  double get_output_offset();

  void set_feedback_offset(double postoffset);
  double get_feedback_offset();
```

```

  void set_tune_in(double value); // enable with break_loop
  double get_tune_in();


  // Operations
  Status calculate_control_cmd(); // calculate next output
  Status init();   // clear time history
  break_loop();    // force tuning inputs
  make_loop();     // enable loop closure
};

// PID Extension
interface PID_TUNING
{
  // Attributes
  double get_Kp();
  double get_Ki();
  double get_Kd();

  void set_Kp(double val);
  void set_Ki(double val);
  void set_Kd(double val);

   double get_Kcommanded();
   double get_kcommanded_dot();
   double get_Kcommanded_dot_dot();
   double get_Kfeedback();

   void set_Kcommanded(double val);
   void set_kcommanded_dot(double val);
   void set_Kcommanded_dot_dot(double val);
   void set_Kfeedback(double val);
};

// Example 1: Software Interfacce to PID Hardware Board
// NULL_CONTROL_AW has same api but does not cause any action
//interface PIDHard: NULL_CONTROL_LAW, PID_TUNING;
// Example 2: Software PID implementation
//interface PIDSoft: CONTROL_LAW, PID_TUNING;
```

4.12 Human Machine Interface

```
interface HMI :
{
  // Presentation Methods
  void present_error_view();
  void present_operational_view();
```

```
  void present_setup_view();
  void present_maintenance_view();

  // Events - to alert HMI that something has happened
  void update_current_view();
};

```

4.13 Process Model

```
// Level 1
interface ProcessModel
{
  OacVector get_user_coordinate_offsets();
  void get_user_coordinate_offsets(OacVector offsets);
  OacVector get_axes_coordinate_offsets();               // used by axes group
  void get_axes_coordinate_offsets(OacVector offsets);   // set by sensor process
  Measure  get_feedrate_override_value();           // used by axisgroup
  void  set_feedrate_override_value(Measure feed);  // used by hmi
  Measure  get_spindle_override_value();            // used by axisgroup
  void  set_spinlde_override_value(Measure feed);   // used by hmi
};

```

4.14 Kinematics

```
// 23-Jun-1997 : Level 1 removed

interface KinStructure
{
  CoordinateFrame get_placement_frame();
  void set_placement_frame(CoordinateFrame value);

  CoordinateFrame get_baseFrame();
  void set_baseFrame(CoordinateFrame value);
};
interface Connection
{
 public:
  KinStructure * get_from();
  void set_from(KinStructure * value);

  KinStructure * get_to();
  void set_to(KinStructure * value);

  CoordinateFrame get_placement();
  void set_placement(CoordinateFrame value);
};
```

```
// FIXME: A template  would map into IDL sequence
typedef RWTPtrSlist<Connection> Connections;

// Last update: 18-Jun-1997 Sushil Birla, Steve Sorensen

interface KinMechanism
{
  void forward_kinematic_transform(Connections &);
  OacVector * inverse_kinematic_transform(CoordinateFrame &)

  Connections get_Connections();
  void set_Connections(Connections value);

  KinMechanisms get_KinMechanisms();
  void set_KinMechanisms(KinMechanisms value);
};

// FIXME: A template  would map into IDL sequence
typedef RWTPtrSlist<KinMechanism> KinMechanisms;

// FIXME: add graph/tree traversal functions

// Notes:
// 1. For various specilizations of inverse_Kinematic_Transform()
// Specialize KinMechanism and extend as needed.
```

4.15 Machine to Machine Interface

This is an outline of the final specification.

```
// MMSDEFS.IDL
// General definitions

    typedef string<32> MMSIdentifier;
// CORBA does not seem to have an identifier type
// The MMSIdentifier consists of a string of characters selected from
//  the letters a-z, the letters A-Z, the digits 0-9, and the underscore
//  character. The MMSIdentifier must begin with an alphabetic character.
//  The MMSidentifier is case sensitive.

// mmssverr.idl
// idl definitions for mms service errors
// see ISO 9506-2:1990 clause 7.5.5 ServiceError

#include mmsspdef.idl
// mms supporting definitions - to get ObjectName

#include mmspidef.idl
```

```
// mms program invocation definitions - to get ProgramInvocationState

#include mmsfidef.idl
// mms file definitions - to get FileSource

    enum VMD_STATE_ERROR_CODE {
      vmd_other,
      vmd_state_conflict,
      vmd_operational_conflict,
      domain_transfer_problem,
      state_machine_id_invalid
    };

    enum APPLICATION_REFERENCE_ERROR_CODE {
      application_reference_other,
      application_unreachable,
      connection_lost,
      application_reference_invalid,
      context_unsupported
    };

    enum DEFINITION_ERROR_CODE {
      definition_other,
      object_undefined,
      invalid_address,
      type_unsupported,
      type_inconsistent,
      object_exists,
      object_attribute_inconsistent
    };

    enum RESOURCE_ERROR_CODE {
      resource_other,
      memory_unavailable,
      processor_resource_unavailable,
      mass_storage_unavailable,
      capability_unavailable,
      capability_unknown
    };

    enum SERVICE_ERROR_CODE {
      service_other,
      primitives_out_of_sequence,
      object_state_conflict,
      service_reserved,
      continuation_invalid,
      object_constraint_conflict
    };
```

```
enum SERVICE_PREEMPT_ERROR_CODE {
  service_preempt_other,
  timeout,
  deadlock,
  cancel
};

enum TIME_RESOLUTION_ERROR_CODE {
  time_resolution_other,
  unsupportable_time_resolution
};

enum ACCESS_ERROR_CODE {
  access_other,
  object_access_unsupported,
  object_non_existent,
  object_access_denied,
  object_invalidated
};

enum INITIATE_ERROR_CODE {
  initiate_other,
  initiate_reserved_1,
  initiate_reserved_2,
  max_services_outstanding_calling_insufficient,
  max_services_outstanding_called_insufficient,
  service_CBB_insufficient,
  nesting_level_insufficient
};

enum CONCLUDE_ERROR_CODE {
  conclude_other,
  further_communication_required
};

enum CANCEL_ERROR_CODE {
  cancel_other,
  invoke_id_unknown,
  cancel_not_possible
};

enum FILE_ERROR_CODE {
  file_other,
  filename_ambiguous,
  file_busy,
  filename_syntax_error,
  content_type_invalid,
```

```
    position_invalid,
    file_access_denied,
    file_non_existent,
    duplicate_filename,
    insufficient_space_in_filestore
  };

  typedef long OTHERS_ERROR_CODE // *** note restriction (long)

  // place holder for companion standard errors
  enum CS_ERROR_CODE {
  };

  enum SERVICE_SPECIFIC_CHOICES {
    obtain_file,
    start,
    stop,
    resume,
    reset,
    deleteVariableAccess,
    deleteNamedVariableAccess,
    deleteNamedVariableList,
    deleteNamedType,
    defineEventEnrollment_Error,
    fileRename,
    additionalService // reserved for companion standards
  };

  union SERVICE_SPECIFIC switch( SERVICE_SPECIFIC_CHOICES ) {
    case obtain_file :    OBTAIN_FILE_ERROR obtain_file_error;
    case start :        START_ERROR start_error;
    case stop :        STOP_ERROR stop_error;
    case resume :        RESUME_ERROR resume_error;
    case reset :        RESET_ERROR reset_error;
    case deleteVariableAccess :
      DELETE_VARIABLE_ACCESS_ERROR delete_variable_access_error;
    case deleteNamedVariableAccess:
  DELETE_NAMED_VARIABLE_ACCESS_ERROR delete_named_variable_access_error;
    case deleteNamedVariableList :
  DELETE_NAMED_VARIABLE_LIST_ERROR delete_named_variable_list_error;
    case deleteNamedType :
      DELETE_NAMED_TYPE_ERROR delete_named_type_error;
    case defineEventEnrollment_Error :
      DEFINE_EVENT_ENROLLMENT_ERROR define_event_enrollment_error;
    case fileRename :    FILE_RENAME_ERROR file_rename_error;
    case additionalService :
      ADDITIONAL_SERVICE_ERROR additional_service_error;
    default :    long dummy; // service specific info not present
```

```
    };


    typedef FileSource OBTAIN_FILE_ERROR;
    typedef ProgramInvocationState START_ERROR;
    typedef ProgramInvocationState STOP_ERROR;
    typedef ProgramInvocationState RESUME_ERROR;
    typedef ProgramInvocationState RESET_ERROR;
    typedef unsigned long DELETE_VARIABLE_ACCESS_ERROR;
    typedef unsigned long DELETE_NAMED_VARIABLE_ACCESS_ERROR;
    typedef unsigned long DELETE_NAMED_VARIABLE_LIST_ERROR;
    typedef unsigned long DELETE_NAMED_TYPE_ERROR;
    typedef ObjectName DEFINE_EVENT_ENROLLMENT_ERROR;
//    enum FILE_RENAME_ERROR { source_file, destination_file };
    typedef FileSource FILE_RENAME_ERROR;
    typedef unsigned long ADDITIONAL_SERVICE_ERROR;
    // The ADDITIONAL_SERVICE_ERROR really should be NULL
    // Can I do that in IDL?

    exception vmd_state_error {
      VMD_STATE_ERROR_CODE    error_code;
      long            additional_code; // note restriction (long)
      string        additional_description;
      SERVICE_SPECIFIC    service_specific;
    };

    exception application_reference_error {
      APPLICATION_REFERENCE_ERROR_CODE    error_code;
      long            additional_code; // note restriction (long)
      string        additional_description;
      SERVICE_SPECIFIC    service_specific;
    };

    exception definition_error {
      DEFINITION_ERROR_CODE    error_code;
      long            additional_code; // note restriction (long)
      string        additional_description;
      SERVICE_SPECIFIC    service_specific;
    };

    exception resource_error {
      RESOURCE_ERROR_CODE    error_code;
      long            additional_code; // note restriction (long)
      string        additional_description;
      SERVICE_SPECIFIC    service_specific;
    };

    exception service_error {
```

```
    SERVICE_ERROR_CODE    error_code;
    long            additional_code; // note restriction (long)
    string        additional_description;
    SERVICE_SPECIFIC    service_specific;
  };

  exception service_preempt_error {
    SERVICE_PREEMPT_ERROR_CODE    error_code;
    long            additional_code; // note restriction (long)
    string        additional_description;
    SERVICE_SPECIFIC    service_specific;
  };

  exception time_resolution_error {
    TIME_RESOLUTION_ERROR_CODE    error_code;
    long            additional_code; // note restriction (long)
    string        additional_description;
    SERVICE_SPECIFIC    service_specific;
  };

  exception access_error {
    ACCESS_ERROR_CODE    error_code;
    long            additional_code; // note restriction (long)
    string        additional_description;
    SERVICE_SPECIFIC    service_specific;
  };

  exception initiate_error {
    INITIATE_ERROR_CODE    error_code;
    long            additional_code; // note restriction (long)
    string        additional_description;
    SERVICE_SPECIFIC    service_specific;
  };

  exception conclude_error {
    CONCLUDE_ERROR_CODE    error_code;
    long            additional_code; // note restriction (long)
    string        additional_description;
    SERVICE_SPECIFIC    service_specific;
  };

  exception cancel_error {
    CANCEL_ERROR_CODE    error_code;
    long            additional_code; // note restriction (long)
    string        additional_description;
    SERVICE_SPECIFIC    service_specific;
  };
```

```
    exception file_error {
      FILE_ERROR_CODE    error_code;
      long             additional_code; // note restriction (long)
      string        additional_description;
      SERVICE_SPECIFIC    service_specific;
    };

    exception others_error {
      OTHERS_ERROR_CODE    error_code;
      long             additional_code; // note restriction (long)
      string        additional_description;
      SERVICE_SPECIFIC    service_specific;
    };

    exception cs_error {
      CS_ERROR_CODE        error_code;
      long             additional_code; // note restriction (long)
      string        additional_description;
      SERVICE_SPECIFIC    service_specific;
    };

#include "mmssverr.idl"
// To get errorcode definitions

// VMDSupport interface
interface VMDSupport {

// methods - Status, UnsolicitedStatus, GetNameList,
//           Identify, GetCapabilityList

// Definitions for Status and UnsolicitedStatus

    enum VMDLogicalStatus {
      STATE_CHANGES_ALLOWED,
      NO_STATE_CHANGES_ALLOWED,
      LIMITED_SERVICES_PERMITTED,
      SUPPORT_SERVICES_ALLOWED
    };

    enum VMDPhysicalStatus {
      OPERATIONAL,
      PARTIALLY_OPERATIONAL,
      INOPERABLE,
      NEEDS_COMMISSIONING
    };

    typedef sequence <boolean,128> VMDLocalDetail;

```

```
    struct Status_Response {
      VMDLogicalStatus    lstat;
      VMDPhysicalStatus    pstat;
      VMDLocalDetail    ldetail;
    };

    typedef boolean Status_Request;

    void Status( in Status_Request req, out Status_Response rsp)
      raises( resource_error, service_error, service_preempt_error,
              access_error, others_error);

    void UnsolicitedStatus( in Status_Response rsp );

// Definitions for GetNameList

    enum VMDObjectClassChoices { VMDOBJECTCLASS, VMDCSOBJECTCLASS };

    enum VMDOBJECTCLASSES { namedVariable, scatteredAccess,
                namedVariableList, nameType, semaphore,
                eventCondition, eventAction, eventEnrollment,
                journal, domain, programInvocation,
                operatorStation };

    union VMDExtendedObjectClass switch(VMDObjectClassChoices) {
      case VMDOBJECTCLASS :        VMDOBJECTCLASSES oc;
      case VMDCSOBJECTCLASS :    int dummy;
    };

    enum VMDObjectScopeChoices {
      vmdSpecific,
      domainSpecific,
      aaSpecific
    };

    union VMDObjectScope switch(VMDObjectScopeChoices)
    {
      case vmdSpecific :    int dummy;
      case domainSpecific:    MMSIdentifier id;
      case aaSpecific:    int dummy;
    };

    struct GetNameList_Request {
      VMDExtendedObjectClass    class;
      VMDObjectScope        scope;
      MMSIdentifier            continueAfter;
    };
```

```
    struct GetNameList_Response {
      sequence <MMSIdentifier>    listOfIdentifier;
      boolean            moreFollows;
    };

    void GetNameList( in GetNameList_Request req,
              out GetNameList_Response rsp)
      raises( vmd_state_error, resource_error, service_error,
          service_preempt_error, access_error, others_error );

// Definitions for Identify

    typedef int Identify_Request;

    struct Identify_Response {
      string<64>    vendorName;
      string<16>    modelName;
      string<16>    revision;
      sequence <MMSObjectIdentifier> listOfAbstractSyntaxes;
    };

    void Identify(in Identify_Request req, out Identify_Response rsp)
      raises( resource_error, service_error, service_preempt_error,
              access_error, others_error );
};

interface VMDSupport; // Corresponds to Remote device verification and probing
interface VariableAccess; // Corresponds to Polled Data Acquisition, Programmed
                          //  data acquistion, and parametric control
interface FileAccess; // No correspondence in IEC 1131-5
interface ResourceManagement; // Corresponds to Application Program Transfer
interface ProgramInvocation; // Corresponds to Program execution and IO control
interface OperatorCommunication; // No correspondence in IEC 1131-5
interface SemaphoreManagement; // Corresponds to Application Program Synch.
interface EventManagement; // Corresponds to Alarm notification
interface Journal Management; // No correspondence in IEC 1131-5

interface VMDSupport {
    void Status(in req, out rsp);
    void UnsolicitedStatus(in req, out rsp);
    void GetNameList( in req, out rsp);
    void Identify(in req, out rsp);
    void GetCapabilityList(in req, out rsp);
};

interface VariableAccess {
    void Read(in req, out rsp);
    void Write(in req, out rsp);
```

```
    void InformationReport(out rsp);
    void GetVariableAccessAttributes(in req, out rsp);
};

interface FileAccess {
    void FileDirectory(in req, out rsp);
    void ObtainFile(in req, out rsp);
    void FileOpen(in req, out rsp);
    void FileRead(in req, out rsp);
    void FileClose(in req, out rsp);
    void FileRename(in req, out rsp);
    void FileDelete(in req, out rsp);
};

interface ResourceManagement {
    void ResourceDownload(in req, out rsp);
    void ResourceUpload(in req, out rsp);
    void RequestResourceDownload(in req, out rsp);
    void RequestResourceUpload(in req, out rsp);
    void ResourceLoad(in req, out rsp);
    void ResourceStore(in req, out rsp);
    void ResourceDelete(in req, out rsp);
    void GetResourceAttributes(in req, out rsp);
};

interface ProgramInvocation {
    void CreateProgramInvocation(in req, out rsp);
    void DeleteProgramInvocation(in req, out rsp);
    void Start(in req, out rsp);
    void Stop(in req, out rsp);
    void Resume(in req, out rsp);
    void Reset(in req, out rsp);
    void Kill(in req, out rsp);
    void GetProgramInvocationAttributes(in req, out rsp);
};

interface OperatorCommunication {
    void input(in req, out rsp);
    void output(in req, out rsp);
};

interface SemaphoreManagement {
    void TakeControl(in req, out rsp);
    void RelinquishControl(in req, out rsp);
    void ReportSemaphoreStatus(in req, out rsp);
    void ReportSempahoreEntryStatus(in req, out rsp);
};
```

```
interface EventManagement {
    void EventNotification(in req, out rsp);
    void AcknowledgeEventNotification(in req, out rsp);
};

interface Journal Management {
    void InitializeJournal(in req, out rsp);
    void WriteJournal(in req, out rsp);
    void ReadJournal(in req, out rsp);
    void GetJournalStatus(in req, out rsp);
};
```

References

[Alb91] J.S. Albus. Outline for a theory of intelligence. *IEEE Transactions on Systems, Man, and Cybernetics*, 21(3), may/june 1991.

[COR91] Object Management Group, Framingham, MA. *Object Management Architecture Guide, Document 92.11.1*, 1991.

[DCO] Distributed Common Object Model.
See Web URL: http://www.microsoft.com/oledev/olemkt/oledcom/dcom95.htm.

[IEC93] International Electrical Commission, IEC, Geneva. *Programmable controllers Part 3 Programming Languages, IEC 1131-3*, 1993.

[Inta] International Organization for Standardization. *ISO 10303-42 Industrial Automation Systems and Integration Product Data Representation and Exchange - Part 42: Integrated Resources: Geometric and Topological Representation.*

[Intb] International Organization for Standardization. *ISO 10303-42 Industrial Automation Systems and Integration Product Data Representation and Exchange - Part 105: Integrated Application Resources: Kinematics.*

[M.S86] M.Shapiro. Structure and Encapsulation in Distributed Systems: The Proxy Principle. In *6th International Conference on Distributed Computing Systems*, pages 198–204. IEEE Computer Society Press, May 1986.

[NGI] Next Generation Inspection System (NGIS).
See Web URL: http://isd.cme.nist.gov/brochure/NGIS.html.

[OMA94] Chrysler, Ford Motor Co. , and General Motors. *Requirements of Open, Modular, Architecture Controllers for Applications in the Automotive Industry*, December 1994. White Paper – Version 1.1.

[OSA96] OSACA. European Open Architecture Effort.
See Web URL: http://www.isw.uni-stuttgart.de/projekte/osaca/english/osaca.htm, 1996.

[PM93] F. M. Proctor and J.L. Michaloski. Enhanced Machine Controller Architecture Overview. Technical Report 5331, National Institute of Standards and Technology, December 1993.

[SOS94] National Center for Manufacturing Sciences. *Next Generation Controller (NGC) Specification for an Open System Architecture Standard (SOSAS)*, August 1994. Revision 2.5.

Table 1: Simplified hypothetical state transition table for LUNL

CurrentState	Event	Action	NextState
LUNL returned	Advance unloader [1]	$obj_1->$execute(...)	UNL advancing
	Advance loader [1]	$obj_2->$execute(...)	L advancing
UNL advancing	Overtime [2]	$obj_3->$execute(...)	Exception
	Unloader is advanced [2]	$obj_4->$execute(...)	UNL advanced
UNL advanced	Return unloader [1]	$obj_5->$execute(...)	UNL returning
UNL returning	Overtime [2]	$obj_6->$execute(...)	Exception
	Unloader is returned	$obj_7->$execute(...)	LUNL returned
L advancing	Overtime [2]	$obj_8->$execute(...)	Exception
	Loader is advanced [2]	$obj_9->$execute(...)	L advanced
L advanced	Return loader [1]	$obj_{10}->$execute(...)	L returning
L returning	Overtime [2]	$obj_{11}->$execute(...)	Exception
	Unloader is returned [2]	$obj_{12}->$execute(...)	LUNL returned
Exception	Reset exception [1]	$obj_{13}->$execute(...)	LUNL returned
...	...	$obj_i->$execute(...)	...

Column explanations:
"LUNL returned" is the starting state, indicating L, UNL are "home".
Event sources indications: [1] MC, [2] DeviceControllerOnoff
Actions include functions of objects in the DeviceControllerOnoff module, as specified in $obj_i->$execute(...) $\forall i$, e.g.,
- get sensed inputs
- check conditions, e.g., "over time limit"
- set actuation outputs
- send LUNL state

Terminology used in the OMAC FSM model:
Transition: The quadruple (event, condition, action, next state);
where "condition" is omitted in this table for simplicity.
CurrentState and NextState are ProcessState objects.
StateRecord: A ProcessState object and its associated Transition objects.
STtable: Container class for a collection of StateRecord objects.